Tierra.doc 17-3-98 documentation for the Tierra Simulator

Tierra Simulator V5.0: Copyright (c) 1991 - 1998 Tom Ray & Virtual Life

Contents

---

1) License Agreement

Tom Ray, ray@udel.edu ray@santafe.edu ray@hip.atr.co.jp (the bulk of the code)
Joseph F. Hart, jhart@hip.atr.co.jp (general programming, Amiga support)
Matt Jones, mjones@condor.psych.ucsb.edu (Mac support)
Agnes Charrel, agnes@sophis.fr, (tping code for network version)
Tsukasa Kimezawa, kim@hip.atr.co.jp (socket code for network version)
Kurt Thearling, KThearling@exapps.com (CM5 adaptation, parallel creatures)
Dan Pirone, cocteau@echannel.com (frontend, crossover)
Tom Uffner, email@unknown (rework of genebanker & assembler)

If you purchased this program on disk, thank you for your support. If you obtained the source code through the net or friends, we invite you to contribute an amount that represents the program's worth to you. You may make a check in US dollars payable to Virtual Life, and mail the check to one of the two listed below.

This is the license agreement:

The source code, documentation, and executables can be freely distributed

The source code and documentation is copyrighted, all rights reserved. The source code, documentation, and the executable files may be freely copied and distributed without fees (contributions welcome), subject to the following restrictions:

- This notice may not be removed or altered.
- You may not try to make money by distributing the package or by using the process that the code creates.
- You may not prevent others from copying it freely.
- You may not distribute modified versions without clearly documenting your changes and notifying the principal author.
- The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
- Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.

The following provisions also apply:

Tom Ray

Zoology Department
University of Oklahoma
Norman, Oklahoma 73019

or

Santa Fe Institute
1399 Hyde Park Road
Santa Fe, NM 87501

ray@santafe.edu (email)
505-984-8800 (Phone)
505-982-0565 (Fax)

or

ATR Human Information Processing Laboratories
2-2 Hikaridai
Seika-cho Soraku-gun
Kyoto 619-02 Japan

ray@hip.atr.co.jp (email)
(81)-7749-5-1063 (Phone)
(81)-7749-5-1008 (Fax)

2) What this Program is, Publications, News

The C source code creates a virtual computer and its operating system, whose architecture has been designed in such a way that the executable machine codes are evolvable. This means that the machine code can be mutated (by flipping bits at random) or recombined (by swapping segments of code between algorithms), and the resulting code remains functional enough of the time for natural (or presumably artificial) selection to be able to improve the code over time.

Along with the C source code which generates the virtual computer, we provide several programs written in the assembler code of the virtual computer. Some of these were written by humans and do nothing more than make copies of themselves in the RAM of the virtual computer. The others evolved from the ancestral programs, and are included to illustrate the power of natural selection.

The virtual machine is an emulation of a MIMD (multiple instruction stream, multiple data stream) shared memory computer. This is a massively parallel computer in which each processor is capable of executing a sequence of operations distinct from the other processors. The parallelism is only emulated by time slicing, but there really are numerous virtual CPUs. One or more CPU will be created and assigned to each ``creature'' (self-replicating algorithm) living in the RAM of the virtual computer. The RAM of the virtual computer is known as the ``soup''.

The operating system of the virtual computer provides memory management and timesharing services. It also provides control for a variety of factors that affect the course of evolution: three kinds of mutation rates, disturbances, the allocation of CPU time to each creature, the size of the soup, the spatial distribution of creatures, etc. In addition, the operating system provides a very elaborate observational system that keeps a record of births and deaths, sequences the code of every creature, and maintains a genebank of successful genomes. The operating system also provides facilities for automating the ecological analysis, that is, for recording the kinds of interactions taking place between creatures.

The version of the software currently being distributed is considered to be a research grade implementation. This means two things: 1) It is under very rapid development, and may not be completely bug free. 2) We have chosen to go with modifiability and modularity over speed of execution.

If you find bugs in the code, please report them to us. By the time you find them and report them, we may have eliminated them, and would be able to provide you with a fixed version. If not, we will be able to fix the bug, and would like to make the fix available to other users.

We have chosen modifiability over speed primarily because we know that the original version of the virtual computer was very poorly designed, except with respect to the features that make it evolvable. Specifically, consider that one third of the original instruction set is taken up by pushing and popping from the stack; there are only two inter-register moves, ax to bx and cx to dx; dx isn't used for anything (in the initial version, dx was used to set the template size, but that has been abandoned); there are no moves between CPU registers and RAM; there is no I/O; and there is no way of addressing a data segment.

In August 1991, 100% of the original virtual CPU code was replaced, with new code that does exactly the same thing. However, the new code is written in a generalized way, a meta-virtual computer, that makes it trivial to alter the machine architecture. With the new implementation of the virtual computer, it is possible for anyone to painlessly swap in their favorite CPU architecture and instruction set, and their innovation will be seamlessly embedded within the heart of the elaborate observational software. Knowing how bad the original design was, there was a temptation to fix it when the virtual computer was reworked, but the original implementation was retained for historical reasons. In spite of its shortcomings, life proliferated in the environment that it created. Things should get interesting as we improve the architecture. The new organization of the code should make that easy.

In August of 1992, the modifications of the previous August bore fruit as three new instruction sets were implemented. These are documented in some detail below, and in general, eliminate most of the problems discussed above.

In early 1994, attempts to create numerous additional new instruction sets revealed that the method of defining instruction sets was still too stiff, so the code used for decoding the instructions was rewritten yet again, providing a much more flexible method of instruction set definition.

The bulk of the code and documentation was written by Tom Ray, whose address is listed at the end of this file. Substantial contributions have been made by: Dan Pirone, cocteau@santafe.edu, was involved in the Tierra project from the Fall of 1990 through 1992, and wrote the user interface. Tom Uffner, tom@genie.slhs.udel.edu, reworked the genebanker and assembler/disassembler in the Fall of 1991. Marc Cygnus, cygnus@udel.edu, developed the ALmond monitor, a separate piece of software that displays activity in a running Tierra (see below). Kurt Thearling has been involved in the port to the CM5, and the development of multi-cellular implementations. Since July 1996, Joe Hart has be programming full-time on the Tierra project.

The behavior of this software is described in the publications which are available on the Tierra home page at:

http://www.hip.atr.co.jp/~ray/pubs/

The Tierra Simulator has been widely reported in the media. These publications are listed on the Tierra home page at: http://www.hip.atr.co.jp/~ray/pubs/news/

---

3) Related Software (Important)

The Tierra simulator is the central piece of a growing set of programs. The accessory programs aid in observing the results of Tierra runs. You will want to select one or both of the following:

3.1) The Beagle Explorer

which graphically displays the output that Tierra saves to disk. Beagle runs only on DOS systems, and while the heart of the source code is available, Beagle uses the Greenleaf DataWindows interface, and that source can not be distributed (available from Greenleaf Software, 16479 Dallas Parkway, Bent Tree Tower Two, Suite 570, Dallas, Texas, 75248, phone: 214-248-2561). Beagle would normally be distributed in the executable form. Once you have the executables, you will be able to run them on your PC, and you will not need Greenleaf. You only need Greenleaf if you plan to modify the source code. Beagle is currently configured only for the CGA or VGA graphics modes. Beagle executables are now available on disk from Virtual Life, or in the same ftp site as the Tierra source code, in the directory: /beagle. If you pick up the Beagle executables from the ftp site, remember to trasfer the files in binary mode.

3.2) The ALmond Monitor

which displays activity in a running Tierra simulator. ALmond runs as a simultaneous but independent process (from Tierra) on the same or on a seperate machine, and establishes network socket communication with Tierra. ALmond can be attached to and detached from a running Tierra simulator without interrupting the simulation. ALmond runs only on unix systems supporting X Windows. The entire ALmond source code is available. Almond was developed and tested on Sun 3 and Sun 4 machines. It was developed under X11R4 by Marc Cygnus. Dan Pirone also did some ALmond development. Question about almond should be directed to Dan Pirone, Marc Cygnus, or Tsukasa Kimezawa.

---

4) Quick Start

The steps required to run the system on DOS and UNIX are slightly different, so there are two sets of instructions listed below.

4.1) DOS Quick Start

If you obtained the Tierra software on disk, the installation program will take care of steps 1 - 3, so you can skip to step 4. If you obtained the software over the net, start with step 1.

step 1) You should have a directory containing the executables and source code and the subdirectories: td, gb0, gb1, gb2, gb3, and gb8. The td directory is where a record of births and deaths will be written. The gb directories contain the initial genomes used to innoculate the soup and the opcode maps. The genebanker will save new genomes to the gb directories. There is a gb directory for each of the four early instruction sets, and two new ones.

step 2) You must compile the assember/disassembler, arg, and the simulator, tierra. We include the two Turbo C V 2.0 project files: tierra.prj and arg.prj. If you are using a more recent version of the compiler, such as Borland C++, you must use the Borland project tool to create a binary project file. Just list the files listed in the two ascii project files that are provided. Compile these projects using the large memory model, make provisions for floating point operations, and link in the graphics library. Put the executables in the path.

step 3) You must assemble the initial genomes, as binaries are not portable. To do this, go into the gb0 directory and type:

```
arg c 0080.gen 80 0080aaa.tie
```

This will create the binary file 0080.gen which contains a creature that you can use to innoculate the soup, the ancestor 0080aaa. You can check to see if this worked by disassembling the genome, by typing:

```
arg x 0080.gen aaa
```

This will create the ascii file 0080aaa. Compare it to the original, 0080aaa.tie (it will not be exactly the same). Before you start a run, copy 0080.gen to 0080gen.vir, in order to have a virgin copy for use later when you start another run.

```
copy 0080.gen 0080gen.vir
```

You can do the same for each of the gb directories (gb0, gb1, gb2, gb3, and gb8). Be sure to assemble the genomes listed at the ends of the corresponding soup_in files (si0, si1, si2, si3, si8).

step 4) Go back to the source code directory and examine the file si0. This file contains all of the parameters that control the run. It is currently set up to innoculate the soup with one cell of genotype 0080aaa, and to run for 500 generations in a soup of 50,000 instructions. You will need a text editor if you want to modify this file. If you use a regular word processor, be sure that you write the file back out as a plain ASCII text file.

step 5) Run the simulator by typing:

```
tierra si0
```

step 6) When the run is over, if you want to start a new run, you should clean up the genebank, because the simulator will read in all genomes in the genebank at startup. The best way to do this is to use the batch files that are provided for this purpose: clr0.bat, clr1.bat, clr2.bat, clr3.bat, and clr8.bat. Since we are discussing a run of instruction set 0, use clr0.bat, by typing clr0 to the DOS prompt. The batch file will take care of the cleanup.

If you wish to use a cumulative genebank in successive runs, use the corresponding cumulative clear batch files: cclr0.bat, cclr1.bat, cclr2.bat, cclr3.bat, and cclr8.bat.

4.2) UNIX Quick Start

step 1) You should have a directory containing the source code and the subdirectories: td, gb0, gb1, gb2, gb3, and gb8. The td (tiedat) directory is where a record of births and deaths will be written. The gb (genebank) directories contain the initial genomes used to innoculate the soup and the opcode map, and the genebanker will save new genomes to these directories.

step 2) You must compile the assember/disassembler, arg, and the simulator, tierra. There is a Makefile included to perform the compilation. This Makefile needs to be edited to comment in the lines for your particular hardware. It has been tested on Sun 3, Sun 4, IBM RS6000, Silicon Graphics Personal Iris and Indigo, DEC DS5000, and NeXT. If you can use the Makefile, type: make, and follow instructions. If all goes well, the Makefile will take care of step 3 for you.

step 3) You must assemble the initial genome, as binaries are not portable. To do this, go into the gb0 directory and type:

```
../arg c 0080.gen 80 0080aaa.tie
```

This will create the binary file 0080.gen which contains a creature that you can use to innoculate the soup, the ancestor 0080aaa. You can check to see if this worked by disassembling the genome, by typing:

```
../arg x 0080.gen aaa
```

This will create the ascii file 0080aaa. Compare it to the original, 0080aaa.tie (they will not be exactly the same). Before you start a run, copy 0080.gen to 0080gen.vir, in order to have virgin copies for use later when you start another run.

```
cp 0080.gen 0080gen.vir
```

You can do the same for each of the gb directories (gb0, gb1, gb2, gb3, and gb8). Be sure to assemble the genomes listed at the ends of the corresponding soup_in files (si0, si1, si2, si3, si8).

step 4) Go back to the source code directory and examine the file si0. This file contains all of the parameters that control the run. It is currently set up to innoculate the soup with one cell of genotype 0080aaa, and to run for 500 generations in a soup of 50,000 instructions.

step 5) Run the simulator by typing: tierra si0 or: tierra si0 > /dev/null & (to run it in the background a Log file can be created by setting the soup_in variable Log = 1)

In order to run tierra in the background, you must compile it with:

```
#define FRONTEND STDIO
```

If you will run Tierra in the foreground, we recommend that you use:

```
#define FRONTEND BASIC
```

These definitions are made in the configur.h file.

step 6) When the run is over, if you want to start a new run, you should clean up the genebank. The best way to do this is to use the Unix script files that have been provided for this purpose (clr0, clr1, clr2, clr3, clr8). You must make the clr# files executable by changing their protection:

```
chmod +x clr0
```

Then all you have to do is type ``clr0'' to the prompt, and the shell script will take care of the cleanup.

If you wish to use a cumulative genebank in successive runs, use the cumulative clear files (cclr0, cclr1, cclr2, cclr3, cclr8). You must also make sure that they are executable:

```
chmod +x cclr0
```

---

5) Running Tierra

This section has the following sub-sections:

5.1) Startup

The first steps in running Tierra are described briefly above. One must place the genomes and the opcode.map in the gb# directory, and one must have created the td directory to receive the output of birth and death data. The genome files are supplied in the form of ASCII assembler code files. These must be assembled into binary form to be able to execute on the virtual machine. If you type arg, the assembler will give you a brief listing of assembler options. More complete documentation of the assembler follows:

5.2) The Assembler/Disassembler

```
     This documentation was written by Tom Uffner.

Arg(1)                   USER COMMANDS                  Arg(1)

NAME
     arg - genbank archive utility

SYNOPSIS
     arg c|r[v12...9] afile size file1 [file2...]
     arg x|t[v12...9] afile [gen1 [gen2...]]

DESCRIPTION
     The arg utility is used to manipulate the genebank  archives
     that  are used by tierra(1).  It is used to assemble or dis-
     sasemble tierra code, list the genomes contained in a  file,
     and also to convert between the old and new file formats.

     The arg commands are:

          c - create afile and add genomes in file1...filen

          r - replace  in  afile  (or  add  to  end)  genomes  in
              file1...filen

          x - extract entire contents or specified  genomes  from
              afile
          t - list entire contents or specified genomes in afile

     The optional modifiers are:

          v - verbose output

          1,2...9 - instruction set (defaults to Format: parameter
                                 of ascii source or to INST = 1)

     Where filen and afile are any legal filenames, genn is  a  3
     character genome label, and size is a decimal integer. (Note
     that tierra(1) expects archives to have names consisting  of
     4 digits, and an extension of .gen, or .tmp.

FILES
     GenebankPath/nnnn.gen permanantly saved genomes
     GenebankPath/nnnn.tmp genomes from periodic saves

SEE ALSO
     An Approach to the Synthesis of Life
     tierra(1), ov(1X), beagle(1DOS), genio(3), arg(5)

BUGS
     Genome extraction and internal  search  functions  could  be
     faster, and will be in the next release.

Tierra, V 4.0      Last change: 8 September 1992                2
```

Please remember that this new form of arg needs the file opcode.map to be in the current working directory. Arg ALWAYS reads the file opcode.map for the mappings for assembling/disassembling genebank archives.

5.3) The Birth-Death Output

During a run, if the DiskOut parameter is non-zero, a record of births and deaths will be written to disk in the path specified by OutPath, to files whose names depend on the BrkupSiz parameter. The format of this file is a bit cryptic, so it will be explained here. The file has either three or four columns of output, depending on whether the GeneBnker parameter is set. Three of the columns remain the same either way: 1) elapsed time since last birth or death event in instructions, output in hexidecimal format. 2) a `b' or a `d' depending on whether this is a birth or a death. 3) the size of the creature in instructions, output in decimal format. If the genebanker is on, then there will be a fourth column containg the three letter code identifying the genotype of the creature. Mutations appear in the birth-death record as the death of one genotype followed by the birth of another, with an elapsed time of zero between the two events.

What makes the file cryptic, and also compact, is that columns are implied to be identical in successive records unless otherwise indicated. Only the first column, elapsed time since last record, must be printed on every line, and only the first record must have all three or four columns. Therefore, if there is a series of successive births, only the first birth record in the series will contain the b. Notice that at the beginning of the file, there will generally be many lines with just one column, because at the outset, all records are of births of the same size and genotype.

The record of births and deaths is read by the Beagle program, and converted into a variety of graphic displays: frequency distributions over time, phase diagrams of the interactions between pairs of sizes or genotypes, or diversity and related measures over time. The source code for reading and interpreting the record of births and deaths is in the bread.c module of the Beagle source code.

5.4) The Genebank Output

If the GeneBnker parameter is set to a non-zero value, then as each creature is born, its genome will be sequenced and compared to that of its mother. If they are identical, the daughter will be assigned the same name as the mother. If they are different, the genome of the daughter will be compared to the same size genomes held in the RAM genebank. If the daughter genome is found in the bank, it will be given the same name as the matching genome in the bank. If the daughter genome is not found in the RAM genebank, it will be compared to any same size genomes stored on the disk that are not in the RAM genebank. If the daughter genome is found in the disk genebank, it will be given the same name as the matching genome in the disk genebank, and that genome will be brougt back into RAM from the disk. If the daughter genome does not match the mother or any genome in either the RAM or disk banks, then it will be assigned an arbitrary but unique three letter code for identification.

The genebanker keeps track of the frequency of each size class and each genotype in the soup. If a genotype exceeds one of the two genotype frequency thresholds, SavThrMem or SavThrPop, its assigned name will be made permanent, and it will be saved to disk in a .gen file. Genotypes are grouped into individual files on the basis of size. For example, all permanent genotypes of size 80 will be stored together in binary form in a file called 0080.gen.

When the simulator comes down, or when the state of the simulator is saved periodically during a run, all genotypes present in the soup which have not been assigned permanent names will be stored in files with a .tmp extension. For example, all temporary genotype names of size 45 would be stored in binary form in a file called 0045.tmp.

The parameter RamBankSiz places a limit on how many genotypes will be stored in RAM. If there are more than RamBankSiz genotypes in the RAM bank, if any of them have permanent genotype names but have gone extinct in the soup, they will be swapped out to disk into a .gen file. Genotypes without permanent names are deleted when they go extinct. If the number of living genotypes in the soup is greater than RamBankSiz, then the value of RamBankSiz is essentially ignored (actually it is treated as if the value were equal to the number of living genotypes). Living genotypes are never swapped out to disk, only extinct ones. Under DOS, the parameter RamBankSiz is of little importance, because it is raised if there are more living genotypes and if the simulator uses all 640K of RAM, RamBankSiz is lowered so that genotypes will be swapped out to avoid running out of memory. Under Unix, this parameter does determine how many genotypes will be held in the rambank, as long as there are not more than RamBankSiz living genotypes.

The binary genebank files can be examined with the assembler-disassembler arg (see the relevant documentation, section 5.2 above). Also, the Beagle Explorer program contains utilities for examing the structure of genomes. One tool condenses the code by pulling out all instructions using templates, which can reveal the pattern of control flow of the algorithm. Another function allows one genome to be used as a probe of another, to compare the similarities and differences between genomes, or to look for the presence of a certain sequence in a genome. A completely separate tool called probe will scan the genebank pulling out any genomes that meet a variety of criteria that the user may define.

5.5) Restarting an Old Run

When you start Tierra by typing tierra to the prompt, you may provide an optional command line argument, which is the name of the file to use as input. This is the file that contains the startup parameters. The default file name is soup_in. When a simulator comes down, and periodically during a run, the complete state of the machine is saved so that the simulator can start up again where it left off. In order to do this you must have the simulator read the file soup_out on startup. This means that you must type:

```
tierra soup_out
```

That is all there is to it.

5.6) The User Interface

There are two interfaces available for Tierra. The source code and executables are shipped in a form that is configured for the nicer of the two, the Basic interface. However, if you make the appropriate modifications to the configur.h file, you can recompile with the Standard Output interface (useful for running Tierra in the background). Documentation for each of these interfaces follows.

5.6.1) The Basic Interface

5.6.1.1) The Basic Screen

The Basic frontend features a dynamic interface to the simulation. The screen area is divided into five basic areas:

The STATS area consists of the top two lines of the screen. This area displays several important variables, whose values are updated after every birth:

```
InstExec  = 75,529993  Cells =  387  Genotypes =  191  Sizes =  23
Extracted = 0080aad @ 8
```

InstExec = 75,529993 tells us that the simulation has executed a total of 75,529,993 instructions. Cells = 387 tells us that there are presently 387 adult cells living in the soup. Genotypes = 191 tells us that there are presently 191 distinct genotypes of adult cells living in the soup. Sizes = 23 tells us that there are presently 23 distinct sizes of adult cells living in the soup. Extracted = 0080aad @ 8 tells us that the last genotype to cross one of the frequency thresholds (SavThrMem or SavThrPop) and get saved to disk and get a permanent name was a creature of size 80 with the name aad, which had a population of 8 adult cells when it crossed the threshold.

The PLAN area displays the values of several variables, whose values are updated every million instructions:

```
InstExeC      =     75  Generations  =    188  Thu Apr 30 11:49:46 1992
    NumCells  =    356  NumGenotypes =    178  NumSizes  =     21
    AvgSize   =     76  NumGenDG     =    171
    AvgPop    =    376  Births       =   1007  Deaths    =   1034
    RateMut   =  10966  RateMovMut   =   1216  RateFlaw  =  97280
    MaxGenPop =     21  (0078aak)    MaxGenMem =     21 (0078aak)
    Speed     =  23256
```

InstExeC = 75 tells us that this set of variables was written when the simulation had executed 75 million instructions. Generations = 188 tells us that the simulation had run for about 188 generations at this time. Thu Apr 30 11:49:46 1992 tells us the actual time and date that this data was printed.

NumCells = 356 tells us that there were 356 adult cells living in the soup. NumGenotypes = 178 tells us that there were 178 distinct genotypes of adult cells living in the soup. NumSizes = 21 tells us that there were 21 distinct sizes of adult cells living in the soup.

AvgSize = 76 tells us that during the last million instructions, the average size was 76. NumGenDG = 171 tells us that there are 171 genotypes that have received permanent names and been saved to disk as .gen files in the genebank directory gb.

AvgPop = 376 tells us that during the last million instructions, the population was 376 adult cells. Births = 1007 tells us that during the last million instructions, there were 1007 births. Deaths = 1034 tells us that during the last million instructions, there were 1034 deaths.

RateMut = 10966 tells us that the actual average background (cosmic ray) mutation rate for the upcoming million instructions will be one mutation per 10966/2 instructions exectued. RateMovMut = 1216 tells us that the actual average move mutation rate (copy error) for the upcoming million instructions will be one mutation for every 1216/2 instructions copied. RateFlaw = 97280 tells us that the actual average flaw rate for the upcoming million instructions will be one flaw for every 97280/2 instructions exectued. The reason that these numbers represent twice the average mutation rates is that they are used to set the range of a uniform random variate determining the interval between mutations.

MaxGenPop = 21 (0078aak) tells us that at this time, the genotype with the largest population is 0078aak, and that it has a population of 21 adult cells. MaxGenMem = 21 (0078aak) tells us that the genotype whose adult cells occupy the largest share of space in the soup is 0078aak, and that it has a population of 21 adult cells.

Speed = 23256 indicates that the virtual cpu is executing 23,256 instructions per second.

The MESSAGE area, for state changes, and Genebank data. This area serves many purposes - memory reallocation messages, Genebank information displays, large interface prompts (eg changing a soup_in variable).

The ERROR area at the second to the last line of the screen. All simulation errors and exit conditions are passed through this area.

The HELP area at the last line of the screen. This area provides suggestions for keystroke navigation. Under DOS it will usually look like this:

```
                 Press any Key for menu ...
```

Under Unix it will generally look like this:

```
                 Press Interupt Key for menu ...
```

Under DOS, pressing any key will get you the menu, under Unix, pressing the interrupt key (usually Ctrl C) will get you the menu, described in the next section.

5.6.1.2) The Menu Options

Please note that if the Tierra simulator is started with two arguments, it will come up with the menu system activated. The first argument must be the name of the soup_in file, the second argument is a dummy and anything will do: tierra soup_in1 junk

The frontend menu looks like this:

```
i-info  v-var  s-save  S-shell  q-save&quit  Q-quit  m-misc  c-continue |->
```

These options allow for rapid IO from the simulation, in a user friendly format. The interface allows transition between any data display mode. The features are ativated by a single keypress ("Enter/Return" is usually not needed).

The options are:

```
  i-info  v-var  s-save  S-shell  q-save&quit  Q-quit  m-misc  c-continue |->
```

i-info: will display some information about creatures stored in the Genebank. See below for details.

v-var: allows you to examine or alter the value of any of the variables in the soup_in file at any point during a run.

s-save: will cause the state of the system to be saved at this point, and then continue the run.

S-shell: this causes you to exit to the DOS prompt, while the simulator remains in the RAM in suspended animation. You can now do anything that doesn't require a lot of memory. Be careful that if you change directories, you come back to the current directories before you exit from the shell. Also, be careful not to do anything that changes the graphics mode.

q-save&quit: will cause the state of the system to be saved at this point, and then exit from the run.

Q-quit: will exit immediately without saving the state of the system.

m-misc: pulls up the miscellaneous sub-menu

c-continue: continue the run.

If you press 'i' for info, from the main TIERRA menu, you are able to select one of the Genebank (GeneBnker = 1) data display modes:

```
INFO   |  p-plan  s-size_histo g-gen_histo m-mem_histo z-size_query     |->
```

The second line from the top of the screen will change, providing additional information on the operating system memory use, or the Tierrian memory use. At the bottom of the screen, a new list of GeneBank data display options are available. All of these modes will try to provide as much data as can fit on the screen. These modes are detailed below:

Hit the c key to continue (to get out of this level of the menu)

The options are:

```
           p-plan  s-size_histo g-gen_histo m-mem_histo z-size_query
```

If you press 'p' from the info menu you get the Plan Display mode:

```
Now in Plan Display mode, updated every million time steps
```

This provides the normal statistics every million virual time steps (this is also an easy way to clear the message area):

```
InstExeC      =     75  Generations  =    188  Thu Apr 30 11:49:46 1992
    NumCells  =    356  NumGenotypes =    178  NumSizes  =     21
    AvgSize   =     76  NumGenDG     =    171
    AvgPop    =    376  Births       =   1007  Deaths    =   1034
    RateMut   =  10966  RateMovMut   =   1216  RateFlaw  =  97280
    MaxGenPop =     21  (0078aak)    MaxGenMem =     21 (0078aak)
    Speed     =  23256
```

The meaning of this information is detailed above.

5.6.1.2.1) The Size Histogram

If you press 's' from the info menu, the message area will show a histogram of frequency distributions of the currently living size classes:

```
   13   1      2 | *
   18   3      9 | **
   19   3      4 | *
   20   2      4 | *
   21   1      3 | *
   22   1      4 | *
   27   3      5 | *
   34   2      4 | *
   35   1      4 | *
   36  34     84 | ****************
   37  70    198 | *************************************
   38  33     92 | ******************
   39   7      9 | **
   40  13     25 | *****
   44   1      2 | *
   45   1      2 | *
   46   2      2 | *
   47   1      4 | *
   49   1      2 | *
   54   1      1 | *
   56   2      3 | *
   70  10     13 | ***
   71  21     42 | ********
   72  24     42 | ********
   73   6      6 | **
   74   7     12 | ***
   75   3      3 | *
   77   2      2 | *
   78   6      7 | **
  109   1      2 | *
```

In the above histogram, the left column of numbers is the size class, the middle column is the number of genotypes of that size class, and the right column is the number of living adult cells of that size class.

5.6.1.2.2) The Memory Histogram

If you press 'm' from the info menu, the message area will show a histogram of frequency distributions of the currently living size classes, by memory use:

```
   Size  Memory use (size * freq.)

   18   3    180 | *
   19   3     76 | *
   20   2     60 | *
   21   1     63 | *
   22   1    110 | *
   27   3    162 | *
   31   1     31 | *
   33   1     33 | *
   34   2    136 | *
   35   1    105 | *
   36  37   3096 | *****************
   37  69   7141 | *************************************
   38  34   3610 | *******************
   39   7    351 | **
   40  13    960 | *****
   41   1     41 | *
   42   1     42 | *
   43   1     43 | *
   44   1     44 | *
   45   1     90 | *
   46   2     92 | *
   47   1    235 | **
   48   1     48 | *
   49   1     98 | *
   50   1     50 | *
   54   1     54 | *
   56   2    168 | *
   66   1     66 | *
   71  22     42 | *
   72  25     42 | *
```

In the above histogram, the left column of numbers is the size class, the middle column is the number of genotypes of that size class, and the right column is the amount of memory occupied by living adult cells of that size class.

5.6.1.2.3) The Genotype Histogram

If you press 'g' from the info menu, the message area will show a histogram of genotype sizes, by specific genotypes:

```
Size      Frequency

    18aab      3 | ***
      aac      6 | ******
    27aab      2 | **
      aad      3 | ***
    35aaa      3 | ***
    36aam      7 | *******
      aat     39 | ***************************************
      abb      3 | ***
    37abb     26 | *************************
      abp      5 | *****
      abt      6 | ******
      acm     23 | **********************
      acn      4 | ****
    38abb      5 | *****
      abc      3 | ***
    40aaj      3 | ***
      aal      2 | **
    47aaa      5 | *****
    49aaa      2 | **
    56aac      2 | **
    70aae      2 | **
      aaf      2 | **
    71aac      3 | ***
      aaj      4 | ****
      aat      3 | ***
      aav      6 | ******
    72aai      2 | **
      aau      3 | ***
      aax      4 | ****
    74aac      4 | ****
```

In the above histogram, the left column of numbers is the size and genotype class, and the right column is the number of living adult cells of that genotype class.

5.6.1.2.4) The Size Class Information Display

If you press 'z' from the info menu, you will be prompted for a specific size class to examine, then you will get a list of the most common genotypes of that size, with some statistics on them:

```
Gene:    #  Mem Errs   Move  Bits
aae      4   0   1     39    EX      TC      TP      MF      MT      MB
aaj      1   0   0      0    EX      TC      TP      MF      MT      MB
aak      1   0   0      0    EX      TC      TP      MF      MT      MB
aan      1   0   0      0    EX      TC      TP      MF      MT      MB
aap      1   0   0      0    EX      TC      TP      MF      MT      MB
aaq      1   0   0      0    EX      TC      TP      MF      MT      MB
aat     10   0   2     38    EXsofh  TCsofh  TPs     MF      MT      MBsof
aau      1   0   0      0    EX      TC      TP      MF      MT      MB
aaw      1   0   0      0    EX      TC      TP      MF      MT      MB
aax      2   0   0      0    EX      TC      TP      MF      MT      MB
```

```
aay      1   0   0      0    EX      TC      TP      MF      MT      MB
aba      1   0   0      0    EX      TC      TP      MF      MT      MB
abb      1   0   0      0    EX      TC      TP      MF      MT      MB
abc      1   0   0      0    EX      TC      TP      MF      MT      MB
abe      1   0   7    146    EX      TC      TP      MF      MT      MB
abi     61   3   2     38    EXsdofh TCsofh  TPs     MFsof   MTsf    MBsdofh
abk      1   0   2     38    EX      TC      TP      MF      MT      MB
abl      1   0   0      0    EX      TC      TP      MF      MT      MB
abm      1   0   0      0    EX      TC      TP      MF      MT      MB
abw     17   1   2     38    EXsdofh TCsofh  TPs     MF      MT      MBsdofh
aby      6   0   1     38    EXsdofh TCsof   TPs     MF      MT      MBsdofh
acb      2   0   0      0    EX      TC      TP      MF      MT      MB
acd      1   0   0      0    EX      TC      TP      MF      MT      MB
ace      1   0   0      0    EX      TC      TP      MF      MT      MB
acf      1   0   0      0    EX      TC      TP      MF      MT      MB
acg      4   0   2     38    EX      TC      TP      MF      MT      MB
ach     13   0   2     38    EXsdofh TCsdofh TPs     MFsdofh MTsdf   MBsdofh
aci      2   0   2     38    EX      TC      TP      MF      MT      MB
acj      2   0   0      0    EX      TC      TP      MF      MT      MB

#    = acutal count, populations of adult cells of this genotype
Mem  = is the percent of Soup, occupied by adult cells of this genotype
Err  = is the number of error flag commited,
Move = number of instructions moved to daughter cell,
Bits = Watch bits, defined in section: 7)  Soup_in Parameters
```

0 values usually represent cases of insufficient data.

If you press 'v' for variable, from the main TIERRA menu, you will get the variable sub-menu, which looks like the following in unix:

```
VAR   |  a - alter variable       e - examine variable
```

From this menu, you have the following options:

a - alter variable: if you press the 'a' key, you can name any variable in the soup_in file, and specify a new value for that variable. Be careful, there are some variables that should not be changed, such as SoupSize.

e - examine variable: if you press the 'e' key, you can examine the value of any variable in the soup_in file.

If you press 'm' for misc, from the main TIERRA menu, you will get the miscellaneous sub-menu, which looks like the following in unix:

```
MISC |  H-Histo Logging  I-Inject Gene  M-Micro Toggle  P-ALmonD Pause  |->
```

and the following in DOS:

```
MISC |  H-Histo Logging  I-Inject Gene M-Micro Toggle    |->
```

When this sub-menu is selected, some information is displayed at the top of the screen about what variables are #defined:

```
VER=4.10 PLOIDY=1 MICRO ALCOMM
```

This information relates to the way that Tierra was compiled. In this case this tells us that this is Version 4.10 of the Tierra program, that it is a haploid model (Ploidy = 1), and that the MICRO virtual debugger is enabled, and that the ALmond communications system is also enabled.

From this menu, you have the following options:

H-Histo Logging: if you press the H key, it will toggle the logging of any histograms you create to the tierra.log file.

I-Inject Gene: inject a genome of your choice from the genebank into the soup of the running simulator.

M-Micro Toggle: turn on the virtual debugger. The debugger has three states: delay, keypress, and off. In delay mode, the debugger will execute one Tierran instruction per second. In the keypress mode, the debugger will execute one Tierran instruction per keypress.

c-continue: exit from this sub-menu.

5.6.1.2.5) The Virtual Debugger

The debugger has three main states: delay, keypress, and off. In delay mode, the debugger will execute one Tierran instruction per second. In the keypress mode, the debugger will execute one Tierran instruction per keypress. Once you have selected your mode, press c to start the simulator again. You can use this tool to step through the genome of a creature, either to see what it does, or to debug a creature that you are writing.

In keypress mode, there are some additional modes. Given that there is usually a population of cells in the soup, the virtual debugger will swap from cell to cell as each one gets its time slice. However, this can be disconcerting if one is trying to study the behavior of a particular cell. The debugger can be made to follow a single creature using the Track cell option as specified at the bottom of the screen in keypress mode:

```
MICRO  |     T-Track cell      U-Track Cpu  ESC-Main Menu  n-Next step
```

If you hit the T key, you will go into Track cell mode, and the menu will change to:

```
MICRO  |   t-Untrack cell      U-Track Cpu  ESC-Main Menu  n-Next step
```

Now hitting the t key will return to the mode that swaps between cells.

Version 4.1 and later of Tierra allows for the possibility that a single cell can have more than one processor. This introduces a similar problem in that the debugger will follow the time slicing pattern of the simulator, which involves a single instruction per processor within a cell. Again, this can make it difficult to follow the activity of a single processor. The debugger can be made to follow a single processor within a cell using the Track Cpu option as specified at the bottom of the screen in keypress mode (see above). If you hit the U key, you will go into Track Cpu mode, and the menu will change to:

```
MICRO  |   t-Untrack cell    u-Untrack Cpu  ESC-Main Menu  n-Next step
```

Please note that going into Track Cpu mode, automatically puts you into Track cell mode. You may exit the Track Cpu mode by hitting the u key, and the menu will change to:

```
MICRO  |   t-Untrack cell      U-Track Cpu  ESC-Main Menu  n-Next step
```

Note that exiting Track Cpu mode does not cause you to exit Track cell mode. If you wish to exit the Track cell mode, you must hit the t key.

If you wish to start debugging from the very beginning of a run, you will want to start the simulator fresh with the menu activated, so that you can start the debugger before the seed creature(s) has started to run. This is done by giving two arguments when starting Tierra, one must be the name of the soup_in file, the second is a dummy argument. So for example, you should type:

```
tierra si0 junk
```

The debugger display in keypress mode looks like:

```
InstExec  = 11,346072  Cells =  679  Genotypes =  274  Sizes =  26
VER=4.0 INST=1 PLOIDY=1    MICRO
 MICRO STEP Mode = keypress

Cell  1: 55 0045aaa @   12595 Slice=  19  Stack[  12628]
IP [  12572]( -23    ) = 0x01     nop1         [  12639]
AX [  30236]                                   [  12595]
BX [  12632]                                   [     45] <
CX [      7]                                   [      0]
DX [      4]                                   [      0]
Flag = 0                                       [      0]
Daughter @ 30199  + 45                         [      0]
                                               [      0]
                                               [      0]
        0 of 1 Cpus

MICRO  |     T-Track cell      U-Track Cpu  ESC-Main Menu  n-Next step
```

The various components of the display are documented in the following:

```
Cell  1: 55 0045aaa @   12595 Slice=  19
```

Each creature has associated with it, a cell structure. The cell structures are organized in a two dimensional array. Cell 1: 55 tells us that the structure for the currently active creature is at location 1,55 in the cells array.

The currently active creature is 0045aaa whose cell starts at address 12595 in the soup. This creature currently has 19 CPU cycles left in its time slice. Note that when the Slice counts down to zero, the slicer will swap in the next creature in the slicer queue, and the debugger will display the CPU of the next creature. Therefore when there are more than one creature in the soup, it is hard to follow the activity of a single creature, since the debugger will be swapped from creature to creature. This problem can be eliminated by activating the Track cell mode (see above).

```
Daughter @ 30199  + 45
```

The currently active creature has allocated a space of 45 instructions at address 30199 in the soup, which it presumably will use to make a daughter.

```
IP [  12572]( -23    ) = 0x01     nop1
```

The instruction pointer (IP) of the currently active creature is presently located at the soup address 12572. Notice that this address is located 23 instruction BEFORE the start of its own cell. The number in parentheses (-23) is the offset of the IP from the start of the cell. As long as the IP is inside the cell, this offset will be displayed without a sign. If the IP is outside of the cell, the offset will be displayed with a + or a - sign. The current offset of -23 means that the instruction pointer of this creature is executing the code of some other creature. In fact 0045aaa is a parasite. The instruction currently being executed is represented in the soup by the hex value 0x01. The assembler mnemonic of this instruction is nop1, it is a no-operation.

```
AX [  30236]
BX [  12632]
CX [      7]
DX [      4]
```

The four CPU registers of the currently active creature contain the values indicated above. Probably, the AX value is the address where the creature is writing in its daughter cell, the BX value is the address that it is copying from in its own cell, the CX value is the number of instructions of the genome remaining to be copied, and the DX value is the size of the last template used.

```
Flag = 0
```

This shows the status of the flag register. This can be used to recognize when an instruction has failed, generating an error condition.

```
Stack[  12628]
     [  12639]
     [  12595]
     [     45] <
     [      0]
     [      0]
     [      0]
     [      0]
     [      0]
     [      0]
```

The stack of the currently active creature contains the values indicated above. The stack pointer points at the location indicated by the < sign (the top of the stack). The values on the stack probably include the address of the instruction pointer (12628) which was pushed on the stack when the parasite called the copy procedure of its host, the address of the end of the creature (12639), the address of the beginning of the creature (12595), and the size of the creature (45).

```
        0 of 1 Cpus
```

Given that version 4.1 allows each cell to have more than one CPU, this message tells us which CPU we are currently observing. The CPUs are numbered starting from 0. In this case, the cell has only one CPU.

5.6.1.2.6) The Genome Injector

The menu system provides a mechanism for injecting genomes into a running simulation. This tool allows a genome from the genebank to be injected into the run at the user's command. The genome must be in a .gen file in the path indicated by the GenebankPath soup_in variable.

If an attempt at injection fails, you will see an error message like:

```
Tierra InjectFromBank() unable to open genome file gb0/0033.gen
                      Tierra ERROR Press any key to continue
```

at the bottom of the screen. If the injection is successful, you will see a message like:

```
 Injection of 0080aaa done
```

at the top of the screen.

There is a function Inject(), in the genebank.c module, which takes a pointer to a genome as an argument. This function can be used to inject genomes from any source. An interesting use of this function would be to facilitate migration of genomes between simulations running on separate machines, creating an archipelago (this is the basis of the new CM5 port, and the new network version of Tierra).

5.6.2) The Standard Output & Interrupt Handler

When Tierra is compiled with: #define FRONTEND STDIO, while running it produces output to the console that looks something like this:

```
TIERRA: LOG = on, Histogram Logging = off
seed = 1573093196
sizeof(Instruction)   = 1
sizeof(TCell)   = 172
sizeof(MemFr) = 16
    60000 bytes allocated for soup
    16512 bytes allocated for cells
    12800 bytes allocated for MemFr
tsetup: arrays allocated without error
skipping 30000 instructions
GetNewSoup: loading 0080aaa into cell 0,2
InstExeC      =        0  Generations  =        0  Wed May 17 16:54:27 1995
    NumCells  =        1  NumGenotypes =        1  NumSizes  =        1
    AvgSize   =       80  NumGenDG     =        1
    RateMut   =     6382  RateMovMut   =     1280  RateFlaw  =    51200
tsetup: soup gotten
                    Press Interrupt Key for Menu ...
extract: 0080aaa @ 268 v
extract: 0080aac @ 11
InstExeC      =        1  Generations  =        3  Wed May 17 16:55:05 1995
    NumCells  =      377  NumGenotypes =      140  NumSizes  =        9
    AvgSize   =       79  NumGenDG     =        2
    AvgPop    =      289  Births       =      988  Deaths    =      612
    Speed     =    26316
    MaxGenPop =      202  (0080aaa)      MaxGenMem =      202 (0080aaa)
    RateMut   =    15211  RateMovMut   =     1264  RateFlaw  =    80000
InstExeC      =        2  Generations  =        5  Wed May 17 16:55:42 1995
    NumCells  =      376  NumGenotypes =      123  NumSizes  =       11
    AvgSize   =       80  NumGenDG     =        2
    AvgPop    =      374  Births       =      880  Deaths    =      881
    Speed     =    27027
    MaxGenPop =      198  (0080aaa)      MaxGenMem =      198 (0080aaa)
    RateMut   =    18105  RateMovMut   =     1280  RateFlaw  =    72640
InstExeC      =        3  Generations  =        8  Wed May 17 16:56:20 1995
    NumCells  =      376  NumGenotypes =      132  NumSizes  =       20
    AvgSize   =       79  NumGenDG     =        2
    AvgPop    =      369  Births       =     1006  Deaths    =     1006
    Speed     =    26316
    MaxGenPop =      164  (0080aaa)      MaxGenMem =      164 (0080aaa)
    RateMut   =    15447  RateMovMut   =     1264  RateFlaw  =    63616
extract: 0080acx @ 10
InstExeC      =        4  Generations  =       10  Wed May 17 16:56:58 1995
    NumCells  =      363  NumGenotypes =      137  NumSizes  =       14
    AvgSize   =       80  NumGenDG     =        3
    AvgPop    =      369  Births       =      852  Deaths    =      865
    Speed     =    26316
    MaxGenPop =      134  (0080aaa)      MaxGenMem =      134 (0080aaa)
    RateMut   =    18305  RateMovMut   =     1280  RateFlaw  =    74496
```

The meaning of each different kind of information is described below:

```
TIERRA: LOG = on, Histogram Logging = off
```

If the soup_in variable ``Log'' is non-zero, most of the information shown in the standard output listing above will be written to the file ``tierra.log'' on disk. Histogram Logging = off indicates that histograms viewed through the menu system will not be saved to the log. This option can be toggled so that static histograms are saved to the log.

```
sizeof(Instruction)   = 1
sizeof(TCell)   = 172
sizeof(MemFr) = 16
```

The size in bytes of each of the main structures, of which the system will allocate large arrays at startup.

```
   60000 bytes allocated for soup
   16512 bytes allocated for cells
   12800 bytes allocated for MemFr
```

The total number of bytes used for each of the three main arrays of structures.

```
tsetup: arrays allocated without error
```

Statement indicating that the arrays were allocated without error.

```
skipping 30000 instructions
```

The soup_in file specified to place the seed creature in the middle of the 60,000 bytes of memory in the soup.

```
seed = 1573093196
```

A record of the seed of the random number generator used in this run. This can be used to repeat the run if desired.

```
GetNewSoup: loading 0080aaa into cell 0,2
```

A statement indicating that the system is innoculating the soup with a creature of size 80. There will be a comparable line for every creature used in innoculating the soup at startup. The first creature goes into cell 2 of array 0, because cells 0 and 1 are used for other purposes.

```
InstExeC      =        0  Generations  =        0  Wed May 17 16:54:27 1995
    NumCells  =        1  NumGenotypes =        1  NumSizes  =        1
    AvgSize   =       80  NumGenDG     =        1
    RateMut   =     6382  RateMovMut   =     1280  RateFlaw  =    51200
tsetup: soup gotten
```

These lines indicate the starting conditions of several variables which will be explained below.

```
extract: 0080aaa @ 268 v
```

This line indicates that the genotype 0080aaa crossed one of the frequency thresholds set in the soup_in file, SavThrMem or SavThrPop, and that there were 268 adult creatures of this genotype in the soup when this was noted. However, no creatures are extracted until the reaper is activated when the soup becomes full. This means that 0080aaa was not actually extracted at the time that it crossed a threshold, but actually much later, when it had a relatively large population. The v after 268 indicates that this was a ``virtual extraction'', which means that the genome was not actually saved to disk, since it already has been saved to disk. Anytime a permanent genotype goes extinct, then reappears and crosses a threshold, it will experience a virtual extraction, which just means that the crossing of the threshold will be reported as an extract in standard out and in the tierra.log file (this information can be put to good use by the tieout tool: tieout tierra.log ie ex)

```
extract: 0080aac @ 11
```

This is a real extraction, as indicated by the absence of a v after the 11. This genotype, 0080aac, crossed the threshold frequency with a population of 11 adult creatures, its name was made permanent, and its genome was saved to disk.

```
InstExeC      =        1  Generations  =        3  Wed May 17 16:55:05 1995
    NumCells  =      377  NumGenotypes =      140  NumSizes  =        9
    AvgSize   =       79  NumGenDG     =        2
    AvgPop    =      289  Births       =      988  Deaths    =      612
    Speed     =    26316
    MaxGenPop =      202  (0080aaa)      MaxGenMem =      202 (0080aaa)
    RateMut   =    15211  RateMovMut   =     1264  RateFlaw  =    80000
```

A statement of this form is printed after every million instructions executed by the system. See the plan() function in the bookeep.c module for more details on this.

```
InstExeC      =        1  Generations  =        3  Wed May 17 16:55:05 1995
```

InstExeC = 1 tells us that one million instructions have been executed in this run. Generations = 3 tells us that roughly three generations of creatures have passed so far during this run. Wed May 17 16:55:05 1995 tells us the time and date of this record.

```
    NumCells  =      377  NumGenotypes =      140  NumSizes  =        9
```

NumCells = 377 tells us that there were 377 adult cells (and a roughly equal number of daughter cells) at this point in the run. NumGenotypes = 140 tells us that there were 140 distinct adult genotypes (code sequences) living in the soup at the time of this record. NumSizes = 9 tells us that there were nine distinct adult genome sizes (creature code lengths) living in the soup at this time.

```
    AvgSize   =       79  NumGenDG     =        2
```

AvgSize = 79 tells us that the average size of all the adult creatures living in the soup at this time was 79 instructions. NumGenDG = 2 tells us that there are two genotypes that has received permanent names and been saved to disk as .gen files in the genebank directory gb.

```
    AvgPop    =      289  Births       =      988  Deaths    =      612
```

AvgPop = 289 tells us that during the last million instructions the average population was 289 adult cells. Births = 988 tells us that during the last million instructions, there were 988 births. Deaths = 612 tells us that during the last million instructions, there were 612 deaths.

```
    MaxGenPop =      202  (0080aaa)      MaxGenMem =      202 (0080aaa)
```

MaxGenPop = 202 (0080aaa) tells us that at this time, the genotype with the largest population is 80aaa, and that it has a population of 202 adult cells. MaxGenMem = 202 (0080aaa) tells us that the genotype whose adult cells occupy the largest share of space in the soup is 80aaa, and that it has a population of 202 adult cells.

```
    RateMut   =    15211  RateMovMut   =     1264  RateFlaw  =    80000
```

RateMut = 15211 tells us that the actual average background (cosmic ray) mutation rate for the upcoming million instructions will be one mutation per 15211/2 instructions exectued. RateMovMut = 1264 tells us that the actual average move mutation rate (copy error) for the upcoming million instructions will be one mutation for every 1264/2 instructions copied. RateFlaw = 80000 tells us that the actual average flaw rate for the upcoming million instructions will be one flaw for every 80000/2 instructions exectued. The reason that these numbers represent twice the average mutation rates is that they are used to set the range of a uniform random variate determining the interval between mutations.

When Tierra is running in the foreground it is possible to interrupt it on either DOS or UNIX, usually by typing Ctrl C (^C). When you do this you will get a brief message listing your options, which looks something like this:

```
^C
=======================================================
 TIERRA: Main Menu
VER=4.20 PLOIDY=1    MICRO
------------------------------------------------------
 InstExe.m    =      4  InstExec.i = 397467  NumCells =  370
 NumGenotypes =    138  NumSizes   =     20
------------------------------------------------------
        Key      Function
        i        Information on simulation
        v        Change or examine a soup_in variable
        m        Misc. commands
        S        Execute a system Shell
        s        Save the soup
        q        Save the soup & quit
        Q        Quit/Abort simulation
        c        To Continue simulation
 ---------------------------------------------------------
i-info  v-var  s-save  S-shell  q-save&quit  Q-quit  m-misc  c-continue |->
```

Unix only: If ALCOMM has been defined at compile time, VPORT is the value of the socket for ALmond data communication.

You must now choose one of the options by typing one of the corresponding letters: ivSsqQc. When you type the letter, the simulator will either prompt you for more input or do the requested operation.

The options are:

```
     i-info v-var s-save S-shell q-save&quit Q-quit c-continue |->
```

i-info: will display some information about creatures stored in the Genebank. See below for details.

v-var: allows you to alter or examine the value of any of the variables in the soup_in file at any point during a run. These options are discussed above in the description of the BASIC frontend.

s-save soup: will cause the state of the system to be saved at this point, and then continue the run.

S-Shell: this causes you to exit to the DOS prompt, while the simulator remains in the RAM in suspended animation. You can now do anything that doesn't require a lot of memory. Be careful that if you change directories, you come back to the current directories before you exit from the shell. Also, be careful not to do anything that changes the graphics mode.

q-save&quit: will cause the state of the system to be saved at this point, and then exit from the run.

Q-quit: will exit immediately without saving the state of the system.

c-continue: continue the run.

i-info: Choosing this option produces a sub-menu:

```
---------------------------------------------------------
        s       Spectrum of all Size Classes
        m       Spectrum of Size Classes, by memory use
        g       Spectrum of Size Classes, by geneotype
        z       Break down of a specific Size Class
                Any other key to main menu ...
---------------------------------------------------------
```

These options are documented under the BASIC frontend above.

5.6.3) The tierra.log file

If the soup_in variable:

```
Log = 1         0 = no log file, 1 = write log file
```

is set to a non-zero value, a file named tierra.log will be written to the current directory. This file contains in abbreviated form, much the same information that is contained in the Standard Output frontend. An example of the output to this file follows:

```
ie0 gn0 Wed May 17 17:27:06 1995
nc1 ng1 ns1
as80 dg1
rm6382 mm1280 rf51200
```

```
ie1 gn3 Wed May 17 17:27:44 1995
nc387 ng141 ns11
as79 dg1
bi1044 de658 ap289
mp211 @ 0080aaa mg211 @ 0080aaa
sp38
rm14287 mm1264 rf75200

ie2 gn5 Wed May 17 17:28:20 1995
nc368 ng128 ns14
as79 dg1
bi847 de866 ap371
mp199 @ 0080aaa mg199 @ 0080aaa
sp36
rm18232 mm1264 rf74688
ex = 0080aab @ 10
ex = 0080aao @ 10

ie3 gn8 Wed May 17 17:28:57 1995
nc376 ng119 ns11
as78 dg3
bi861 de853 ap376
mp179 @ 0080aaa mg179 @ 0080aaa
sp37
rm18226 mm1248 rf74624
ex = 0080abr @ 10

ie4 gn10 Wed May 17 17:29:34 1995
nc380 ng117 ns12
as78 dg4
bi892 de888 ap372
mp164 @ 0080aaa mg164 @ 0080aaa
sp37
rm17387 mm1248 rf71872
ex = 0045aaa @ 14
```

Because this file is of essentially the same form as the standard output, only the abbreviations will be documented here. Refer to the documentation of the Standard Output to interpret the meaning of this file.

```
ie = InstExeC; gn = Generations; nc = NumCells; ng = NumGenotypes;
ns = NumSizes; as = AvgSize; dg = NumGenDG; bi = Births; de = Deaths;
ap = AvgPop; mp = MaxGenPop; mg = MaxGenMem; sp = Speed
rm = RateMut; mm = RateMovMut; rf = RateFlaw;
ex = extract
```

6) Listing of Distribution Files

The distribution includes the following files:

announce.XX - previous announcements about Tierra releases and other Tierra news.

arg.c - the main module for the assembler/disassembler, originally written by Tom Uffner. This program converts ascii assembler files into binary files which can be executed by the Tierran virtual computer.

arg.prj - the Turbo C V 2.0 project file for compiling the assember/disassembler. If you are using a more recent Borland compiler, you must use the Borland tool to build the binary project file. Include the modules listed in this ascii version of the file.

arg_inc.c - a file used to include some of the files used in compiling the assembler (arg). They are included in this way because arg_inc.c defines ARG, which alters the included modules to be compatible with arg rather than tierra.

alcomm.doc - some comments on the socket communications mechanisms used by the almond monitor.

arginst.h - a file containing some variable used by the assembler.

beagle.doc - documentation for the Beagle Explorer program, a DOS based tool for examining the results of running Tierra, after the fact.

bookeep.c - source code for bookeeping routines, which keep track of how many of what kind of creatures are in the soup, and other stuff like that.

cclr0, cclr1, cclr2, cclr3, cclr0.bat, cclr1.bat, cclr2.bat, cclr3.bat - Unix shell scripts, and DOS batch files for cleaning up an old run in preparation for a new run with a cumulative genebank (or you can just use it to remove files generated by a run, to get you disk space back). You must change the protection on the unix files to make them executable: chmod +x cclr0 There is a different file for each instruction set.

clr0, clr1, clr2, clr3, clr0.bat, clr1.bat, clr2.bat, clr3.bat - Unix shell scripts and DOS batch files for cleaning up an old run in preparation for a new run (or you can just use it to remove files generated by a run, to get you disk space back). You must change the protection on the unix files to make them executable: chmod +x clr0 There is a different file for each instruction set.

configur.h - a file for configuring Tierra. You probably won't need to touch this unless you get into advanced stuff.

create.tex, create.ps - This is the text of a manuscript which has been published in French translation in the September 1992 issue of a popular Swiss magazine named LeTemps.

declare.h - all global variables are declared in this file, except those whose values are set by soup_in. Those globals are declared in soup_in.h. declare.h is included by tierra.c which contains the main function.

decode.c - the decode functions interpret the executable code of the creatures, and map it onto the executable functions contained in the instruct.c module.

diskbank.c - some functions that deal with writing to disk.

extern.h - all global variables are delcared as extern in this file, and this file is included by all *.c files except tierra.c which includes delcare.h instead.

frontend.c - functions for handling input/output for Tierra. This stuff was written by Dan Pirone. This module contains basic data reporting functions, and includes one of the following: tstdio.c - low level IO for standard IO, simple, and good for debugging. tcurses.c - advanced IO for unix, based on "standard" curses calls, Please see the manual pages on curses for the nitty gritty details. tturbo.c - advanced IO for the Turbo C / DOS environment. On machines with EGA or VGA harware, 43/50 lines modes are used.

genio.c - functions for input/output of creatures. This stuff is also used by arg.c, the assembler/disassembler. This module has benefited from a lot of work by Tom Uffner.

instruct.c - this module contains generalized executable functions. These generalized functions are mapped to specific functions by the parsing functions in the parse.c module.

license.h - a file stating the terms of the license agreement, which is include in all the source code modules.

Makefile, MakeAlmond, MakeCM5 - the make files, for compiling tierra and arg under Unix. MakeAlmond compiles Tierra for use with ALmond, and MakeCM5 compiles to run Tierra on the CM5.

memalloc.c - functions for handling memory allocation in the soup, the stuff that ``cell membranes'' are made of.

memtree.c - additional memory allocation routines, written by Chris Stephenson of the IBM T. J. Watson Research Center.

mlayer.doc - some comments on the ALmond tool.

multi_cell.ps, multi_cell.ps.gz - a manuscript describing some initial experiments with evolving multi-cellular digital organisms.

OV_README.txt - some documentation on the overview (ALmond) tool.

PhysicaD.ps, PhysicaD.tex - a manuscript describing the results of three studies with Tierra: a comparison of the patterns of evolution in four instruction sets, the relationship between evolution and entropy, and the evolution of complexity.

portable.c - functions for portability between operating systems.

portable.h - definitions for portability between operating systems and architectures.

prototyp.h - all functions in Tierra are prototyped here.

queues.c - queue management functions for the slicer and reaper queues.

quickstart - the quickstart section of this document, yanked out and made as a separate file, for those who don't read the doc files.

rambank.c - functions for managing the genebank. This module has benefited from a lot of work by Tom Uffner.

reserves.tex - a document describing a proposal to create a network-wide biodiversity reserve for digital organisms.

slicers.c - interchangeable slicer functions. This file contains some experiments in the allocation of CPU time to creatures. This is an interesting thing to play with.

si0, si1, si2, si3 - the ascii files read by Tierra on startup, which contains all the global parameters that determine the environment. There is a different file for each instruction set.

soup_in.h - this file defines the default values of all the soup_in variables, and defines the instruction set by mapping the assember mnemonics to the opcodes, decode functions, and executables.

tcurses.c - see under frontend.c above.

tcolors.cfg - an ascii file that maps the colors of the Tierra display in DOS. This makes it easy to remap the colors. This is useful for odd displays such as LCD where the colors do not map nicely.

thoughts.ps, thoughts.tex - This is the text of a thought experiment. It was written before the Tierra program started running. After Tierra ran, this manuscript was slightly edited, but it has not been altered since February 2, 1990. This manuscript shows the thought processes that led to the development of the Tierra program.

tieout.c - a separate tool for observing the contents of the tierra.log file documented in tieout.doc.

tieout.doc - documentation of the tieout tool.

tieout.exe - the DOS distribution also includes the executable version of the tieout tool, so that it does not have to be compiled. See tieout.doc.

tierra.c - this file contains the main function, and the central code driving the virtual computer.

tierra.doc - this file, on line documentation.

tierra.exe - the DOS distribution includes the executable version of the Tierra program, so that it does not have to be compiled.

tierra.h - this file contains all the structure definitions. It is a good source of documentation for anyone trying to understand the code.

tierra.prj - the Turbo C V 2.0 project file for compiling Tierra. If you are using a more recent Borland compiler, you must use the Borland tool to build the binary project file. Include the modules listed in this ascii version of the file.

tierra.tex - This is the text of a manuscript detailing the Tierra program and the early results from its use. This manuscript is published as a working paper of the Santa Fe Institute.

tmonitor.c - this file contains support calls for the ALmond tool.

trand.c - random number generation routines from Numerical Recipes in C.

tsetup.c - routines called when Tierra starts up and comes down. Tom Uffner has put some work into this module as well.

ttools.c - routins used in generating the histograms displayed by the frontend.

Zen.ps, Zen.tex - a manuscript describing a way of thinking about using natural evolution to create artificial life.

gb0: - a subdirectory containing the genomes of the creatures saved during a run of instruction set 0.

gb0/0080aaa.tie - the ancestor, written by a human, mother of all other creatures.

gb0/0073aaa.tie - a new ancestor, written by a human, like 0080aaa, but with some junk code removed.

gb0/0022aaa.tie - the smallest non-parasitic self-replicating creature to evolve.

gb0/0045aaa.tie - the archtypical parasite.

gb0/0046aaa.tie - a symbiont with 0064aaa. These two were created by hand, by splitting 0080aaa into two parts.

gb0/0064aaa.tie - a symbiont with 0046aaa. These two were created by hand, by splitting 0080aaa into two parts.

gb0/0072aaa.tie - a phenomenal example of optimization through evolution, involving the unrolling of the copy loop.

At the time of this writing, only a single run, of a billion instructions, has been analyzed at the level of detail to reveal the specifics of the ecological arms race that took place. That information is presented in the manuscripts cited above. This run can be seen on the video, available from Media Magic, P.O. Box 507, Nicasio, CA 94946. Some of the major players in that run are listed below, and included in the software distribution, in the chronological order of their appearance:

gb0/0079aab.tie - this creature dominated the soup at the time that a major optimization took place, resulting in 0069aaa and its relatives.

gb0/0069aaa.tie - this fully self-replicating (non-parasitic) creature appeared at a time that the dominant self-replicating size class was 79. Thus this creature was a hopeful monster. It managed to shave 10 instructions off of the genome in a single genetic change. Actually this creature claims that its immediate ancestor (parental genotype) was 0085aal. If this is true, then it actually shaved 16 instructions off its size in a single genetic event. Unfortunately, 0085aal was not preserved in the fossil record.

gb0/0069aab.tie - this host creature co-dominates the soup for a time, with the parasite 0031aaa.

gb0/0031aaa.tie - this parasite co-dominates the soup for a time, with the host 0069aab.

gb0/0070aaw.tie - while 0069aab and 0031aaa co-dominate the soup, this creature, which is the first hyper-parasite, surges up. It and its relatives drive the parasites to extinction.

gb0/0061aag.tie - this hyper-parasites dominates at the time that the first social creatures appear.

gb0/0061aai.tie - this is the first social hyper-parasite to surge up. It is social by virtue of using a template in its tail to jump back to its head. This only works when it occurs in close aggregations of same kind creatures.

gb0/0061aab.tie - this is the second social hyper-parasite to surge up. It operates on the basis of a completely different social mechanism from 0061aai. In this creature the basis of cooperation is that the template used to search for its tail, does not match to the tail template. However, it two of these creatures abut in memory, the union of the tail template of one with the head template of the next, forms the template that is used to identify the tail. This algorithm also only works when the creatures occur in aggregations. However, this mechanism is more fragile than that used by 0061aai. For 0061aab, the two cooperating creatures must exactly abut. For 0061aai, the two cooperating creatures may have some space between them. Therefore it is not surprising that the social mechanism of 0061aai is the one prevailed.

gb0/0061aaa.tie - this is the social hyper-parasite that dominated at the time that cheaters invaded.

gb0/0027aab.tie - this is the dominant cheater that invaded against the social hyper-parasites.

gb0/0080.gen - the DOS distribution also includes the assembled binary version of the 0080aaa.tie file, so that Tierra can be run without having to assemble the genome file first.

gb0/arg.exe - the DOS distribution includes the executable version of the assembler/disassembler.

gb0/opcode.map - a file read by Tierra, Beagle, and arg upon startup. This file determines the correspondence between opcodes and the associated machine instructions. This is a simple way to play with the definition of the physics and chemistry of the simulator.

gb0/opcode.vir - a virginal copy of opcode.map. You may use the rnd_inst tool to create new opcode maps, or you may create them by hand. You will have to name the new versions opcode.map in order to assemble the genomes based on the new mapping. You can keep opcode.vir in reserve.

gb0/probe.doc - documentation of the probe tool.

gb0/probe.c - a separate tool for observing the genebank, documented in probe.doc.

gb0/probe.exe - the DOS distribution includes the executable version of the probe program, so that it does not have to be compiled.

gb0/rnd_inst.c - source code of the run_inst tool.

gb0/rnd_inst.doc - documentation of the run_inst tool.

gb0/rnd_inst.exe - the DOS distribution includes the executable version of the rnd_inst program, so that it does not have to be compiled.

gb1: - a subdirectory containing the genomes of the creatures written for use with instruction set 1.

gb1/0005aaa.tie - non-replicating program written to test get().

gb1/0010aaa.tie - non-replicating program written to test put.

gb1/0011aaa.tie - non-replicating program written to test get with template.

gb1/0013aaa.tie - non-replicating program to test put with template.

gb1/0016aaa.tie - non-replicating program to test put.

gb1/0060aaa.tie - a parasite hand-made from the ancestor.

gb1/0095aaa.tie - the ancestor for use in instruction set 1.

gb1/opcode.map - the opcode map for use in instruction set 1.

gb1/opcode.vir - a virginal copy of the opcode map for use in instruction set 1.

gb2: - a subdirectory containing the genomes of the creatures written for use with instruction set 2.

gb2/0050aaa.tie - a parasite hand-made from the ancestor.

gb2/0093aaa.tie - the ancestor for use in instruction set 2.

gb2/opcode.map - the opcode map for use in instruction set 2.

gb2/opcode.vir - a virginal copy of the opcode map for use in instruction set 2.

gb3: - a subdirectory containing the genomes of the creatures written for use with instruction set 3.

gb3/0082aaa.tie - the ancestor for use in instruction set 3.

gb3/opcode.map - the opcode map for use in instruction set 3.

gb3/opcode.vir - a virginal copy of the opcode map for use in instruction set 3.

td: - a subdirectory where a complete record of births and deaths will be written.

td/break.1 - a file containing a record of births and deaths. A dummy file is provided to hold the space.

td/diverse.c - a separate tool for observing the diversity of genotypes and size classes in a run, documented in diverse.doc.

td/diverse.doc - documentation of the diversity tool.

td/diverse.exe - the DOS distribution also includes the executable version of the diversity index tool, so that it does not have to be compiled. See diverse.doc.

td/fragment.c - a separate tool for peparing fragments of a run for observation by the beagle tool, documented in beagle.doc.

td/fragment.exe - the DOS distribution also includes the executable version of the fragment tool. See beagle.doc.

td/run_info.c - a separate tool for peparing the output of a run for observation by the beagle tool, documented in beagle.doc.

td/run_info.exe - the DOS distribution also includes the executable version of the run_info tool. See beagle.doc.

7) soup_in Parameters

A typical soup_in file looks like the following:

```
/* begin soup_in file */

# tierra core: 17-5-95 INST == 0

# observational parameters:

BrkupSiz = 1024      size of output file in K, named break.1, break.2 ...
CumGeneBnk = 1    Use cumulative gene files, or overwrite
debug = 0               0 = off, 1 = on, printf statements for debugging
DiskBank = 1    turn disk-genebanker on and off
DiskOut = 1             output data to disk (1 = on, 0 = off)
GeneBnker = 1   turn genebanker on and off
GenebankPath = gb0/  path for genebanker output
hangup = 0            0 = exit on error, 1 = hangup on error for debugging
Log = 1         0 = no log file, 1 = write log file
MaxFreeBlocks = 800     initial number of structures for memory allocation
OutPath = td/  path for data output
SaveFreq = 10      frequency of saving core_out, soup_out and list
SavRenewMem = 0   free and renew dynamic memory after saving to disk
SavMinNum = 10     minimum number of individuals to save genotype
SavThrMem = .02  threshold memory occupancy to save genotype
SavThrPop = .02  threshold population proportion to save genotype
WatchExe = 1     mark executed instructions in genome in genebank
WatchMov = 1     set mov bits in genome in genebank
WatchTem = 1     set template bits in genome in genebank

#NET
# LocalPort = 8001   Port number for local node
# map_fn = MapFile   mapfile for Internet
# Nice = 19          Nice value for cpu
# MonPort = 9001     Monitor tool port number

# environmental variables:

alive = 990000      how many generations will we run
DistFreq = -.3     frequency of disturbance, factor of recovery time
DistProp = .2      proportion of population affected by distrubance
DivSameGen = 0 cells must produce offspring of same genotype, to stop evolution
DivSameSiz = 0 cells must produce offspring of same size, to stop size change
DropDead = 5 stop system if no reproduction in the last x million instructions
EjectRate = 50 rate at which random ejections from soup occur
GenPerBkgMut = 16 mutation rate control by generations ("cosmic ray")
GenPerFlaw = 32       flaw control by generations
GenPerMovMut = 8    mutation rate control by generations (copy mutation)
IMapFile = opcode.map map of opcodes to instructions, file in GenebankPath
JmpSouTra = 0.   source track switches per average size
JumpTrackProb = .2 probability of switching track during a jump of the IP
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (parse dependant)
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
MalSamSiz = 0  force memory alloc to be same size as parent (stop evolution)
```

```
MalTol = 20 multiple of avgsize to search for free block
MateProb = 0.0  probability of mating at each mal
MateSearchL = 5 multiple of avgsize to search 0 = no limit
MateSizeEp = 2 size epsilon for potential mate
MateXoverProp = 1.0 proportion of gene to secect for crossover point
MaxCpuPerCell = 16   maximum number of CPUs allowed per cell
MaxIOBufSiz = 8      maximum size for IOS buffer
MaxMalMult = 3        multiple of cell size allowed for mal()
MaxPutBufSiz = 4      maximum size for put IO buffer
MaxGetBufSiz = 4      maximum size for get IO buffer
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  rwx protection for memory owned by a creature
MemModeProt = 2  rwx protection for memory owned by another creature
# rwx protect mem: 1 bit = execute, 2 bit = write, 4 bit = read
MinCellSize = 12       minimum size for cells
MinTemplSize = 1       minimum size for templates
MovPropThrDiv = .7       minimum proportion of daughter cell filled by mov
new_soup = 1          1 = this a new soup, 0 = restarting an old run
NumCells = 6      number of creatures and gaps used to inoculate new soup
PhotonPow = 1.5         power for photon match slice size
PhotonWidth = 8   amount by which photons slide to find best fit
PhotonWord = chlorophill  word used to define photon
PutLimit = 10 distance for intercellular communication, mult of avg creat siz
ReapRndProp = .3 top prop of reaper que to reap from
SearchLimit = 5 distance for template matching, mult of avg creat siz
seed = 0 seed for random number generator, 0 uses time to set seed
SizDepSlice = 0  set slice size by size of creature
SlicePow = 1    set power for slice size, use when SizDepSlice = 1
SliceSize = 25    slice size when SizDepSlice = 0
SliceStyle = 2   choose style of determining slice size
SlicFixFrac = 0   fixed fraction of slice size
SlicRanFrac = 2   random fraction of slice size
SoupSize = 60000 size of soup in instructions

space 10000
0080aaa
space 10000
0045aaa
space 10000
0080aaa

/* end soup_in file */
```

The ordering of parameters is not important, except that the creatures used to innoculate the soup must be listed at the end of the file, with no blank lines between the creatures (if there is more than one). Parameters that are not listed in the soup_in or soup_out files, default to the values set in the soup_in.h file. The parameter names must begin at the first character of the line (no spaces or other characters may appear at the beginning of the line before the parameter name).

The meaning of each of these parameters is explained below:

```
# tierra core: 14-12-93 INST == 0

# observational parameters:
```

These lines are comments. The code that reads the soup_in and soup_out files skips all blank lines, and all lines beginning with the # character. The input parameters have been divided into two groups, the ``observational parameters'' and the ``environmental variables''. The observational parameters are listed first, and they do not affect the course of events in the run. They only affect what kind of information we extract, and how and where it is stored.

```
BrkupSiz = 1024      size of output file in K, named break.1, break.2 ...
```

If this value is set to zero (0) the record of births and deaths will be written to a single file named tierra.run. However, if BrkupSiz has a non-zero value, birth and death records will be written to a series of files with the names break.1, break.2, etc. Each of these files will have the size specified, in K (1024 bytes). The value 1024 indicates that the break files will each be one megabytes in size. The output file(s) will be in the path specified by OutPath (see below). See also DiskOut. Please note that these files can get quite large.

```
CumGeneBnk = 1  0 = start new genebank, 1 = use cumulative genebank
```

This parameter gives us the option of starting a fresh genebank, or of working with a cumulative genebank. If you are planning to compare the results of a series of runs, you should consider using a cumulative genebank. With a cumulative genebank, if the identical genotypes appear in successive runs, they will have identical names.

```
debug = 0               0 = off, 1 = on, printf statements for debugging
```

This is used during code development, to turn on and off print statements for debugging purposes.

```
DiskBank = 1    turn disk-genebanker on and off
```

It is optional to save successful genomes to disk. This parameter determines if genomes are saved to disk.

```
DiskOut = 1            output data to disk (1 = on, 0 = off)
```

If this parameter is set to zero (0), no birth and death records will be saved. Any other value will cause birth and death records to be saved to a file whose name is discussed under BrkupSiz above, in the path discussed under OutPath below.

```
GeneBnker = 1   turn genebanker on and off
```

The parameter turns the genebanker on and off. The value zero turns the genebanker off, any other value turns it on. With the genebanker off, the record of births and deaths will contain the sizes of the creatures, but not their genotypes. Also no genomes will be saved in the genebank. When the genebanker is turned on, the record of births and deaths will contain a three letter unique name for each genotype, as well as the size of the creatures. Also, any genome whose frequency exceeds the thresholds SavThrMem and SavThrPop (see below) will be saved to the genebank, in the path indicated by GenebankPath (see below).

```
GenebankPath = gb0/  path for genebanker output
```

This is a string variable which describes the path to the genebank where the genomes will be saved. The path name should be terminated by a forward slash.

```
hangup = 0            0 = exit on error, 1 = hangup on error for debugging
```

If an error occurs which is serious enough to bring down the system, having hangup set to 1 will prevent the program from exiting. In this case, the program will hang in a simple loop so that it remains active for debugging purposes. Set this parameter only if you are running the simulator under a debugger.

```
Log = 1         0 = no log file, 1 = write log file
```

If Log is non zero, a disk file called tierra.log will be written mirroring all major simulation states. If disk space is a problem, please set Log = 0. This can be changed at any point in the run.

```
MaxFreeBlocks = 700     initial number of structures for memory allocation
```

There is an array of structures used for the virtual memory allocator. This parameter sets the initial size of the allocated array, at startup.

```
OutPath = td/  path for data output
```

The record of births and deaths will be written to files in a directory specified by OutPath. See BrkupSiz above for a discussion of the name of the file(s) containing the birth and death records.

```
SaveFreq = 100        frequency of saving core_out, soup_out and list
```

Every SaveFreq million instructions, the complete state of the virtual machine is saved. This is a useful feature for long runs, so that the system can be restarted if it is interrupted for some reason.

```
SavRenewMem = 0   free and renew dynamic memory after saving to disk
```

When this variable is set to 1, at the time of SaveFreq, all allocated memory is freed after being written to disk. Then the allocated memory is reallocated and the data is read in from disk. The objective of this is to eliminate memory fragmentation. However, this is a new variable and still needs some debugging. It has bad interactions with some other variables. It is advised not to set this variable to 1 for now.

```
SavMinNum = 10     minimum number of individuals to save genotype
```

A genotype will not be given a permanent name and saved to disk in a .gen file unless it has a population of SavMinNum or more adult individuals.

```
SavThrMem = .02  threshold memory occupancy to save genotype
SavThrPop = .02  threshold population proportion to save genotype
```

These two variables affect the rate at which genomes are assigned permanent names and saved to disk. In a soup of 60,000 instructions (SoupSize = 60000), thresholds of .05 will save about one genome per two million instructions, thresholds of .04 will save about one genome per million instructions, thresholds of .03 will save about 1.2 genomes per million instructions, thresholds of .02 will save about three genome per million instructions. On DOS systems, care should be taken to avoid saving too many genomes as they will clog the memory and bring the system down (See KNOWN BUGS below).

```
SavThrMem = .04  threshold memory occupancy to save genotype
```

If a particular genotype fills SavThrMem of the total space available in the soup, it will be assigned a permanent unique name, and saved to disk. Note that an adjustment is made because only adult cells are counted, and embryos generally fill half the soup. Therefore adult cells of a particular genotype need only occupy SavThrMem * 0.5 of the space to be saved.

```
SavThrPop = .04  threshold population proportion to save genotype
```

If a particular genotype amounts to SavThrPop of the total population of (adult) cells in the soup, it will be assigned a permanent unique name, and saved to disk.

```
WatchExe = 0     mark executed instructions in genome in genebank
WatchMov = 0     set mov bits in genome in genebank
WatchTem = 0     set template bits in genome in genebank
```

WARNING: setting any of these three watch parameters will engage CPU intensive observational software that will substantially slow down any simulation. Engage these tools only if you plan to use the data that they generate.

```
WatchExe = 0     mark executed instructions in genome in genebank
```

If the genebank is on, setting this parameter to a non-zero value will turn on a watch of which instructions are being executed in each permanent genotype (this helps to distinguish junk code from code that is executed), and also, who is executing whose instructions. There is a bit field in the GList structure (bit definitions are defined in the tierra.h module) that keeps track of whether a creature executes its own instructions, those of another creature, if another creature executes this creatures instructions, etc:

```
    bit  2  EXs = executes own instructions (self)
    bit  3  EXd = executes daughter's instructions
    bit  4  EXo = executes other cell's instructions
    bit  5  EXf = executes instructions in free memory
    bit  6  EXh = own instructions are executed by other creature (host)
```

```
WatchMov = 0     set mov bits in genome in genebank
```

If the genebank is on, setting this parameter to a non-zero value will turn on a watch of who moves whose instructions and where. This information is recorded in the bit field in GList structure:

```
    bit 17  MFs = moves instruction from self
    bit 18  MFd = moves instruction from daughter
    bit 19  MFo = moves instruction from other cell
    bit 20  MFf = moves instruction from free memory
    bit 21  MFh = own instructions are moved by other creature (host)
    bit 22  MTs = moves instruction to self
    bit 23  MTd = moves instruction to daughter
    bit 24  MTo = moves instruction to other cell
    bit 25  MTf = moves instruction to free memory
    bit 26  MTh = is written on by another creature (host)
    bit 27  MBs = executing other creature's code, moves inst from self
    bit 28  MBd = executing other creature's code, moves inst from daughter
    bit 29  MBo = executing other creature's code, moves inst from other cell
    bit 30  MBf = executing other creature's code, moves inst from free memory
    bit 31  MBh = other creature uses another CPU to move your instructions
```

```
WatchTem = 0     set template bits in genome in genebank
```

If the genebank is on, setting this parameter to a non-zero value will turn on a watch of whose templates are matched by whom. This information is recorded in the bit field in the GList structure:

```
    bit  7  TCs = matches template complement of self
    bit  8  TCd = matches template complement of daughter
    bit  9  TCo = matches template complement of other
    bit 10  TCf = matches template complement of free memory
    bit 11  TCh = own template complement is matched by other creature (host)
    bit 12  TPs = uses template pattern of self
    bit 13  TPd = uses template pattern of daughter
    bit 14  TPo = uses template pattern of other
    bit 15  TPf = uses template pattern of free memory
    bit 16  TPh = own template pattern is used by other creature (host)
```

```
# LocalPort = 8001   Port number for local node
# map_fn = MapFile   mapfile for Internet
# Nice = 19          Nice value for cpu
# MonPort = 9001     Monitor tool port number
```

These varibles are only for the network version.

```
# environmental variables:
```

This is a comment indicating the beginning of the environmental parameters. These parameters affect the course of evolution and ecology of the run.

```
alive = 500            how many generations will we run
```

This tells the simulator how long to run, in generations.

```
DistFreq = -.3       frequency of disturbance, factor of recovery time
```

The frequency of disturbance, as a factor of recovery time. This and the next option control the pattern of disturbance. If you do not want the system to be disturbed, set DistFreq to a negative value. If DistFreq has a non-negative value, when the soup fills up the reaper will be invoked to kill cells until it has freed a proportion DistProp of the soup. The system will then keep track of the time it takes for the creatures to recover from the disturbance by filling the soup again. Let's call this recovery time: rtime. The next disturbance will occur: (rtime X DistFreq) after recovery is complete. Therefore, if DistFreq = 0, each disturbance will occur immediately after recovery is complete. If DistFreq = 1, the time between disturbances will be twice the recovery time, that is, the soup will remain full for a period equal to the recovery time, before another disturbance hits. Using this parameter (positive values) will probably create problems in conjuction with memory allocation modes that specify the address where the daughter will be placed.

```
DistProp = .2       proportion of population affected by distrubance
```

The proportion of the soup that is freed of cells by each disturbance. The disturbance occurs by invoking the reaper to kill cells until the total amount of free memory is greater than or equal to: (DistProp X SoupSize). Note that cells are not killed at random, they are killed off the top of the reaper queue, except as modified by the ReapRndProp variable.

```
DivSameGen = 0 must produce offspring of same genotype, to stop evolution
```

This causes attempts at cell division to abort if the offspring is of a genotype different from the parent. This can be used when the mutation rates are set to zero, to prevent sex from causing evolution.

```
DivSameSiz = 0 cells must produce offspring of same size, to stop size change
```

Like DivSameGen, but cell division aborts only if the offspring is of a different size than the parent. Changes in genotype are not prevented, only changes in size are prevented.

```
DropDead = 5 stop system if no reproduction in the last x million instructions
```

Sometimes the soup dies, such as when mutation rates are too high. This parameter watches the time elapsed since the last cell division, and brings the system down if it is greater than DropDead million instructions.

```
EjectRate = 50 rate at which random ejections from soup occur
```

Controls the rate at which creatures being born into the soup are ejected from the soup. The value 50 means one in fifty creatures. In a single processor implementation, ejection means death. However, on a parallel machine like the CM5, or in a networked implementation, ejection means migration to a soup on another processor.

```
GenPerBkgMut = 16 mutation rate control by generations ("cosmic ray")
```

Control of the background mutation rate ("cosmic ray"). The value 16 indicates that in each generation, roughly one in sixteen cells will be hit by a mutation. These mutations occur completely at random, and also affect free space where there are no cells. If the value of GenPerBkgMut were 0.5, it would mean that in each generation, each cell would be hit by roughly two mutations.

```
GenPerFlaw = 64       flaw control by generations
```

Control of the flaw rate. The value 64 means that in each generation, roughly one in sixty-four individuals will experience a flaw. Flaws cause instructions to produce results that are in error by plus or minus one, in some sense. If the value of GenPerFlaw were 0.5, it would mean that in each generation, each cell would be hit by roughly two flaws. This parameter has a profound effect on the rate at which creatures shrink in size under selection for small size.

```
GenPerMovMut = 8    mutation rate control by generations (copy mutation)
```

Control of the move mutation rate (copy mutation). The value 8 indicates that in each generation, roughly one in eight cells will be hit by a mutation. These mutations only affect copies of instructions made during replication (by the double indirect mov instruction). When an instruction is affected by a mutation, one of its five bits is selected at random and flipped. If the value of GenPerMovMut were 0.5, it would mean that in each generation, each cell would be hit by roughly two move mutations.

```
IMapFile = opcode.map map of opcodes to instructions, file in GenebankPath
```

Names the file containing the Instruction Set Map. The simulator will look for this file in the GenebankPath. Please remember to only use genebank files created with compatible map files. This is facilitated by having each different opcode.map file in a separate genebank directory.

```
JmpSouTra = 0.   source track switches per average size
JumpTrackProb = .2 probability of switching track during a jump of the IP
```

These are experimental parameters that have been introduced to control track expression in diploid (or higher) implementation. Each CPU has a source and destination track defined, four use with the move instructions. The movii instruction copies from the source track to the destination track. The source track will be randomly changed at the rate defined by the JmpSouTra variable. Each CPU also has an execute track defined. This will also change at random during any jump of the instruciton pointer, at a probability determined by the JumpTrackProb variable.

```
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (parse dependant)
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
MalTol = 20 multiple of avgsize to search for free block
```

Memory allocation control:

Chris Stephenson of the IBM T. J. Watson Research Center (cjs@yktem.vnet.ibm.com) has contributed a new memory allocator to Tierra, based on a cartesian tree scheme that he developed. The new allocator is more efficient than the old and provides a variety of options for controlling where offspring are placed. Free blocks are maintained in a two dimensional binary tree, in which daughter blocks can not be larger than their parents, and blocks are ordered from left to right according to their position in memory.

```
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (decode dependant)
```

This variable provides seven options for controlling where offspring are placed in the soup.

0 = first fit: this is the method used in versions of Tierra before V4.0. In this method, free blocks are checked starting with the leftmost block in memory, and the first block large enough to accomodate the request us used.

1 = better fit: this method starts from the root of the tree, and if the free block is larger than the request, it moves to the daughter free block that is closest to the size of the request, this provides a better fit of the request to the free block, but not necessarily the best fit.

2 = random preference: blocks of memory are allocated at positions selected at random from the soup.

3 = near mother's address: blocks of memory will be placed near the creature making the request (daughters will be placed near mothers), within the specified tolerance (see MalReapTol and MalTol).

4 = near dx address: blocks of memory will be placed near the address specified in the dx register, within the specified tolerance (see MalReapTol and MalTol).

5 = near top of stack address: blocks of memory will be placed near the address specified at the top of the stack, within the specified tolerance (see MalReapTol and MalTol).

6 = suggested address (decode dependant): this is effectively like modes 4 and 5, except that the source of the requested address is variable, and will be decided by the decode code. In other words, the target address could be specified by the value in any register.

```
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
```

When a position and tolerance is specified for the daughter cell, the reaper will probably have to free memory in that area to service the request. If the reaper kills strictly by the reaper queue, without regard to location, it may kill many cells before killing in the vicinity of the request. This creates situations in which it is difficult for populations to grow, because the reaper kills much more than is necessary. To resolve this problem, an option is added in which the reaper still uses the queue, but it only kills cells that fall within the tolerance region of the requested location. The variable MalReapTol turns this option on.

```
MalSamSiz = 0 force memory alloc to be same size as parent (stop evolution)
```

Setting this variable to 1 will cause all memory allocations by creature to be the same size as the creature. This contributes to preventing evolution.

```
MalTol = 20 multiple of avgsize to search for free block
```

This determines a region of tolerance within which a request for a specific location of memory can be serviced. It is specified as a multiple of the average creature size.

```
MateProb = 0.2  probability of mating at each mal
MateSearchL = 5 multiple of avgsize to search 0 = no limit
MateSizeEp = 2 size epsilon for potential mate
MateXoverProp = 1.0 proportion of gene to secect for crossover point
```

Haploid Sex (aka crosover):

Haploid sex has been implemented in mimicry of Walter Tackett's (tackett@ipld01.hac.com) implementation in his private version of Tierra. This form of sex is more in the spirit of the genetic algorithm that Tierra, in that the creatures don't have any choice in the matter, they are forced to have sex at god's whim. However, it does have the virtue of creating something like a biological species, in that individuals close in size exchange genes creating an evolving gene pool.

The Tierran reproduction mechanism of repeatedly copying data from an offset in the mother cell to an offset (presumably) in a daughter cell, can be modified to easily allow ``crossover'' of genetic data. Crossover seems to provide a means of increasing the rate of evolution. It can be thought of as a higher mutation rate which can ``favor good genes''. This affect will be made clear by euclidating how haploid crossover ``sex'' is done in the Tierran model.

At the time a creature allocates a block of memory for a daughter (mal), Tierra decides if this creature will mate, using the soup_in variable MateProb (0.0 = no mating at all, 100.0 = creatures will try to mate 100% of the time). If the creature is not selected for mating, Tierra continues as normal until the next time this creature executes a memory allocate. If this creature does mate, a point in the creatures genome, selected randomly from the proportion MateXoverProp of the genome size, is chosen as the ``cross over'' point. Also whether we use the first part, or the last part of the genome is also chosen (chances are 1 in 2 for each "half"). No further actions are taken until the creature begins moving data (presumably from herself, to the daughter cell).

At the time that the creature first copies an instruction of its genome to a new location (presumably in the daughter cell) using the moviab instruction, a mate is chosen. The mate is chosen at this point because the chances of the mate dying before the cross over is complete are smaller now than at the time of the memory allocation. A potential mate must be similar in size to the creature trying to mate (creature's size +/- MateSizeEp), and may not be of the exact same genotype (eg. 0080aaa can't mate with another 0080aaa, there would be no point in the operation). The search radius from the creature to the mate is determinied by MateSearchL. MateSearchL is multiplied by the Average Creature Size to determine the number of bytes to search. NOTE - in release 3.13, only FORWARD (towards higher addresses) in done. This will become bidirectional in future releases. Even if a creature is selected for mating a the time of execution of the mal instruction, it may not actually mate, because it may not be able to find a mate of a different genotype, in the correct size range, and within the search radius.

If we have chosen to contribute our first section to the daughter, our mate will contribute the second section. In that case, the copying of the genome will proceed normally until we reach the crossover point. After that, each time we execute the moviab instruction, we will copy an instruction from the genome of our mate rather than from our genome. The creature is ``unaware'' of this process. For the sake of clarity an example is given:

Example A - Mate contributes second half :

```
  creature ( 10 bytes long ) & mate ( 9 bytes long )
  [ step 0 ] mal - selects Xover point of 5, second half
  [ step 1 ] move bytes at offset 0-4 from self
  [ step 2 ] move byte at offset 5-9 from mate
  [ step 3 ] move byte after last byte of mate ( 10th byte )
```

Example B - Mate contributes first half :

```
  creature ( 10 bytes long ) & mate ( 11 bytes long )
  [ step 0 ] mal - selects Xover point of 5, first half
  [ step 1 ] move bytes at offset 0-4 from mate
  [ step 2 ] move byte at offset 5-10 from self
```

```
MaxCpuPerCell = 16   maximum number of CPUs allowed per cell
```

Beginning with version 4.1, it is possible for individual cells to have more than one CPU, to allow parallelism. The maximum number of CPUs that a single cell may have is determined by this variable.

```
MaxMalMult = 3        multiple of cell size allowed for mal()
```

When a cell attempts to allocate a second block of memory (presumably to copy its genome into), this parameter is checked. If the amount of memory requested is greater than MaxMalMult times the size of the mother cell, the request will fail. This prevents mutants from requesting the entire soup, which would invoke the reaper to cause a massive kill off.

```
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  rwx protection for memory owned by a creature
MemModeProt = 2  rwx protection for memory owned by another creature
```

From the point of view of a particular creature, memory is divided into three classes: owned by a creature, owned by another creature, not owned by any creature. Each of these three classes of memory may have a unique read, write, execute memory protection for the creature in question. 1 bit = execute, 2 bit = write, 4 bit = read So:

	execute	write	read
0	Y	Y	Y
1	N	Y	Y
2	Y	N	Y
3	N	N	Y
4	Y	Y	N
5	N	Y	N
6	Y	N	N
7	N	N	N

Be aware that each of the three forms of memory protection is turned on at compile time by the appropriate definitions:

```
#define READPROT     /* define to implement read protection of soup */
#define WRITEPROT    /* define to implement write protection of soup */
#define EXECPROT     /* define to implement execute protection of soup */
```

The version distributed on disk only has write protection turned on. To implement the other forms of protection you will need to comment in the appropriate line in the configur.h file, and recompile. Once the relevant form of protection is defined at compile time, the MemModexxxx variables still determines how it is actually used. We do this because each form of memory protection is costly of CPU time, and write protection is the only one likely to be used.

```
MemModeFree = 0  read, write, execute protection for free memory
```

When memory is free, not owned by any creature, it will have this protection. 0 means that the creature can read, write, or execute the free memory.

```
MemModeMine = 0  rwx protection for memory owned by a creature
```

This determines how the creature may use the memory that it owns. 0 means that the creature can read, write, or execute its own memory.

```
MemModeProt = 2  rwx protection for memory owned by another creature
```

When memory is owned by a creature other than the one that is currently executing, it will have this protection. 2 means that write privelage is protected, so only the owner can write on the memory. The creature that is currently executing will not be able to write on this memory.

```
MinCellSize = 8       minimum size for cells
```

When a cell attempts to divide, this parameter is checked. If the daughter cell would be smaller than MinCellSize instructions, divide will fail. The reason this is needed is that with no lower limit, there is a tendency for some mutants to spawn large numbers of very small cells.

```
MinTemplSize = 1       minimum size for templates
```

When an instruction (like jump) attempts to use a template, this parameter is checked. If the actual template is smaller than MinTemplSize instructions, the instruction will fail. This is a matter of taste.

```
MovPropThrDiv = .7       minimum proportion of daughter cell filled by mov
```

When a cell attempts to divide, this parameter is checked. If the mother cell has moved less than MovPropThrDiv times the mother cell size, of instructions into the daughter cell, cell division will abort. A value of .7 means that the mother must at least fill the daughter 70% with instructions (though all these instructions could have been moved to the same spot in the daughter cell). The reason this parameter exists is that without it, mutants will attempt to spew out large numbers of empty cells.

```
new_soup = 1          1 = this a new soup, 0 = restarting an old run
```

This value is checked on startup, to determine if this is a new soup, or if this is restarting an old run where it left off. When the system comes down, all soup_in parameter (and many other global variables) are saved in a file called soup_out. The value of new_soup is set to 0 in soup_out. In order to restart an old run, just use soup_out as the input file rather than soup_in. This is done by using soup_out as a command line parameter at startup: tierra soup_out

```
NumCells = 3      number of creatures and gaps used to inoculate new soup
```

This parameter is checked at startup, and the system will look for a list of NumCells creatures at the end of the soup_in file. The value 3 indicates that the soup will initially be innoculated by three cells. However, NumCells also counts gaps that are placed between cells (without gaps, all cells are packed together at the bottom of the soup at startup). Notice that after the list of parameters in the soup_in file, there is a blank line, followed by a list of genotypes. The system will read the first NumCells genotypes from the list, and place them in the soup in the same order that they occur in the list.

```
PhotonPow = 1.5         power for photon match slice size
```

If SliceStyle (see below) is set to the value 1, then the allocation of CPU cycles to creatures is based on a photon - chlorophyll metaphor. Imagine that photons are raining down on the soup at random. The cell hit by the photon gets a time slice that is proportional to the goodness of fit between the pattern of instructions that are hit, and an arbitrary pattern (defined by PhotonWord, see below).

The template of instructions defined by PhotonWord is laid over the sequence of instructions at the site hit by the photon. The number of instructions that match between the two is used to determine the slice size. However, the number of matching instructions is raised to the power PhotonPow, to calculate the slice size.

```
PhotonWidth = 8   amount by which photons slide to find best fit
```

When a photon hits the soup, it slides a distance PhotonWidth, counting the number of matching characters at each position, and the slice size will be equal to the number of characters in the best match (raised to the power PhotonPow, see above). If PhotonWidth equals 8, the center of the template will start 4 instructions to the left of the site hit by the photon, and slide to 4 instructions to the right of the site hit.

```
PhotonWord = chlorophill  word used to define photon
```

This string determines the arbitrary pattern that absorbs the photon. It uses a base 32 numbering system: the digits 0-9 followed by the characters a-v. The characters w, x, y and z are not allowed (that is why chlorophyll is misspelled). The string may be any length up to 79 characters.

```
PutLimit = 10 distance for intercellular communication, mult of avg creat siz
```

The put/get instructions allow messages to be passed between cells. In one mode, the addressing of the target cell is based on complementary template matching. The distance that is searched to find complementary templates is determined by this variable.

```
ReapRndProp = .3 top prop of reaper que to reap from
```

This parameter determines the degree to which mortality is random. If ReapRndProp is set to zero, the reaper always kills the creature at the top of the reaper queue. If ReapRndProp is set to one, the reaper kills at random. If ReapRndProp is set to 0.3, the reaper will kill a cell selected at random from the top 30% of the reaper queue.

```
SearchLimit = 5
```

This parameter controls how far instructions may search to match templates. The value five means that search is limited to five times the average adult cell size. The actual distance is updated every million instructions.

```
seed = 0 seed for random number generator, 0 uses time to set seed
```

The seed for the random number generator. If you use the value zero, the system clock combined with the random number generator is used to set the seed. If you use any other value, it will be the seed. The starting seed (even when provided by the clock) will be written to standard output, the tierra.log file, and also saved in the soup_out file when the simulator comes down. By using the original seed and all the same initial parameter settings in soup_in, a run may be repeated exactly.

```
SizDepSlice = 0  set slice size by size of creature
```

This determines a major slicer option. If this parameter is set to zero, the slice size will either be a constant set by SliceSize (see below) or a uniform random variate, or a mix of the two. The mix is determined by the relative values of SlicFixFrac and SlicRanFrac (see below). The actual slice size will be:

```
(SlicFixFrac * SliceSize) + (tlrand() % (I32s) ((SlicRanFrac * SliceSize) + 1))
```

If SizDepSlice is set to a non-zero value, the slice size will be proportional to the size of the genome. In this case, the base slice size will be the genome size raised to the power SlicePow (see below). To clarify let slic_siz = genome_size ^ SlicePow, the actual slice size will be:

```
(SlicFixFrac * slic_siz) + (tlrand() % (I32s) ((SlicRanFrac * slic_siz) + 1))
```

```
SlicePow = 1    set power for slice size, use when SizDepSlice = 1
```

This parameter is only used when SizDepSlice = 1. In this case, the genome size is raised to the power SlicePow to determine the slice size (see algorithm under SizDepSlice above). If SlicePow = 1, the run will be size neutral, selection will not be biased toward either large or small creatures (the probability of an instruction being executed is not dependent on the size of the genome it is located in). If SlicePow > 1, selection will favor larger genomes. If SlicePow < 1, selection will favor small genomes.

```
SliceSize = 25    slice size when SizDepSlice = 0
```

This parameter determines the base slice size when SizDepSlice = 0. The actual slice size in this case depends on the values of SlicFixFrac and SlicRanFrac (see below). The way the slice size is actually calculated is explained under SizDepSlice above.

```
SliceStyle = 2   choose style of determining slice size
```

The slicer is a pointer to function, and the function actually used is determined by this parameter. At present there are three choices (0-2). The pointer to function is assigned in the setup.c module, and the slicer functions themselves are contained in the slicers.c module. 0 = SlicerQueue() - slice sizes without a random component 1 = SlicerPhoton() - slice size based on photon interception metaphor 2 = RanSlicerQueue() - slice size with a fixed and a random component

```
SlicFixFrac = 0   fixed fraction of slice size
```

When SliceStyle = 2, the slice size has a fixed component and a random component. This parameter determines the fixed component as a multiple of SliceSize, or genome_size ^ SlicePow.

```
SlicRanFrac = 2   random fraction of slice size
```

When SliceStyle = 2, the slice size has a fixed component and a random component. This parameter determines the random component as a multiple of SliceSize, or genome_size ^ SlicePow.

```
SoupSize = 50000 size of soup in instructions
```

This variable sets the size of the soup, measured in instructions.

```
space 10000
0080aaa
space 10000
0045aaa
space 10000
0080aaa
```

This is the list of cells that will be loaded into the soup when the simulator starts up. This example indicates that three cells will be loaded at startup, the ancestor 0080aaa alternating with the parasite 0045aaa. The line:

```
space 10000
```

indicates that a space of ten thousand instructions will be left before the cell that follows. Therefore the first cell will start after 10,000 blank instructions, and there will be 10,000 instructions between each of the three creatures. Please note that the parameter NumCells must match the number of creatures/spaces to be read in (see NumCells above).

---

8) The Instruction Sets

8.0) The CPU

Many aspects of the instruction sets can be best understood if we also understand the structure of the CPU. Therefore we will first review all the details of the current CPU structure.

The CPU is associated with the cell, and so it structure hangs off the cell structure:

```
struct cell {   /* structure for cell of organisms */
    Dem d;  /* fecundity and times and dates of birth and death */
    Que q;  /* pointers to previous and next cells in queues */
    Mem mm; /* main cell memory */
    Mem md; /* daughter cell memory */
    CpuA c; /* virtual cpus */
    I8s ld; /* 0 = dead, 1 = alive */
} ;     /* sizeof(struct cell) = XX */
```

The CpuA structure is:

```
typedef struct { /* structure for cpu array */
    I16s ib; /* instruction bank */
    I32s n;  /* number of allocated cpus */
    I32s ac; /* number of this active cpu */
    I8s  sync; /* sync flag for this cell */
    Cpu  *c; /* pointer to currently active cpu */
    Cpu  *ar; /* pointer to array of cpus */
    InstDef *d; /* pointer to current InstDef structure for parsing */
#ifdef NET
    IOb io;              /* IO buffer for network communications */
#endif /* NET */
#ifdef IO
    PGb pg;              /* IO buffer for put - get communications */
    GloCom gc;
#endif /* IO */
} CpuA;
```

When a CPU receives a time slice, its instruction bank is incremented by the slice size:

```
    ce->c.ib += size_slice;
```

As the slice is executed, the instruction bank is decremented, and the slice exits when the slice decrements to zero:

```
    for(is.ts = ce->c.ib; is.ts > 0; )
    {   ...
        ce->c.ib -= is.dib;
        is.ts -= is.dib;
    }
```

To allow for multi-threaded (multi-cellular) function, each CpuA structure can accomodate any number of CPUs by virtue of having an array of CPUs pointed to by the ar element:

```
    Cpu  *ar; /* pointer to array of cpus */
```

The number of CPUs actually allocated is indicated by:

```
    I32s n;  /* number of allocated cpus */
```

The currently active CPU is pointed to by:

```
    Cpu  *c; /* pointer to currently active cpu */
```

and its position in the ar array is indicated by:

```
    I32s ac; /* number of this active cpu */
```

When an instruction is decoded, a pointer to the InstDef structure for the decoded instruction is set:

```
    InstDef *d; /* pointer to current InstDef structure for parsing */

I16s FetchDecode()
{   I16s    di;

    is.eins = &soup[ce->c.c->ip];
    di = (*(is.eins)) % InstNum;
    ce->c.d = id + di;    <---- Set pointer to InstDef structure
    (*id[di].decode)();
    ...
```

When a cell has multiple CPUs, it is possible to synchronize them by using:

```
    I8s  sync; /* sync flag for this cell */
```

in the CpuA structure, combined with:

```
    I8s sync;            /* wait for sync */
```

in the Cpu structure. This is done with the use of the csync() instruction. This will cause each CPU to wait until all the CPUs have executed the csync() instruction. This is done to line up the CPUs to some desired relative configurations.

For the network version, each cell has one IO buffer for communications:

```
    IOb io;              /* IO buffer for network communications */
```

This buffer is shared by all CPUs of the cell. This buffer is manipulated by the following functions in the instruct.c file:

```
IOS * GetFreeIOS(cp) /* find an IOS to write incoming message to */
IOS * GetIOSOut(cp) /* get address of IOS output structure */
IOS * GetTagIOS(cp, tag) /* find next IOS with specified tag, to read */
void IncrIOSnio(cp) /* increment cp->c.io.nio with funky modulus */
I8s ReadIOS(cp, tag, dest) /* copy data from IOS buffer to soup */
I8s WriteIOS(cp, sou) /* copy data from soup to IOS buffer */
void InitIOS(ios) /* initialize IOS structure */
```

There are also some non-network communications buffers that allow communications between cells within a soup:

```
    PGb pg;             /* IO buffer for put - get communications */
    GloCom gc;
```

This buffer is manipulated by the following functions in the instruct.c file:

```
Preg GetFreeGetBuf(cp) /* find get buf reg to write incoming message to */
Preg GetFreePutBuf(cp) /* find put buf reg to write incoming message to */
void put() /* write a value to the output buffer (prayer) */
void puticc() /* write a value to the output buffer */
I8s ReadFPut(cp, value) /* for god to read the data in the output buffer */
void get() /* read a value from the input buffer */
void Write2Get(cp, value) /* place value in input buffer of cell cp */
void Broad2Get(value) /* broadcast value to input buffer of all cells */
```

The ``Cpu *c; /* pointer to currently active cpu */'' element points to the currently active CPU, which is a Cpu structure:

```
typedef struct {        /* structure for registers of virtual cpu */
    Reg re[ALOC_REG];   /* array of registers */
    Reg ip;             /* instruction pointer */
    Reg sp;             /* stack pointer */
    Reg st[STACK_SIZE]; /* stack */
    Flags fl;           /* flags */
    CRflags cf;         /* CPU Register flags */
    I8s sync;           /* wait for sync */
#if PLOIDY > 1
    I8s ex; /* track of execution */
    I8s so; /* source trace for reads */
    I8s de; /* destination track for writes */
    I8s wc; /* wait count for error-track switching */
#endif /* PLOIDY > 1 */
} Cpu;
```

```
    Reg re[ALOC_REG];   /* array of registers */
```

are the actual registers of the CPU. There are obviously ALOC_REG registers in the CPU, though if shadow referencing is used, there are half that number of registers available for computation. The instruction pointer is:

```
    Reg ip;             /* instruction pointer */
```

There is a stack:

```
    Reg st[STACK_SIZE]; /* stack */
```

with STACK_SIZE words, and a stack pointer:

```
    Reg sp;             /* stack pointer */
```

In the case of multiple CPUs per cell, the element:

```
    I8s sync;           /* wait for sync */
```

is used for synchronizing all the CPUs of the cell to some positions in their code, using the csync() instruction.

There are some additional elements used for ploidy levels greater than 1:

```
    I8s ex; /* track of execution */
    I8s so; /* source trace for reads */
    I8s de; /* destination track for writes */
    I8s wc; /* wait count for error-track switching */
```

These are experimental elements used to control the expression of the multiple copies of the code, the switching between execution tracks. This control has not been completely worked out.

There are two sets of flags for the CPU. One of these:

```
    Flags fl;            /* flags */
```

uses the Flags structure:

```
typedef struct {
    int  E:  1; /* Error : 1 = error condition has occurred
                0 = no error condition occurred */
    int  S:  1; /* Sign : resulting sign of arithmetic operation,
                0 = +, 1 = - */
    int  Z:  1; /* Zero : result of arithmetic or compare operation,
                0 = nonzero, 1 = zero */
    int  B:  2; /* Bits: 00 = 32, 01 = 16, 10 = 8 */
    int  D:  1; /* Direction: for shift, rotate, move, search */
} Flags;
```

These flags are all cleared by the clrf() instruction.

The:

```
    int  E:  1; /* Error : 1 = error condition has occurred
                0 = no error condition occurred */
```

flag is set by an error condition. The:

```
    int  S:  1; /* Sign : resulting sign of arithmetic operation,
                0 = +, 1 = - */
```

flag is set when a computation yields a negative value. The:

```
    int  Z:  1; /* Zero : result of arithmetic or compare operation,
                0 = nonzero, 1 = zero */
```

flag is set when an arithmetic operation yields a zero result, or when a comparison yields a true value.

The intention is that the E, S, and Z flags be used by conditional instructions. The conditional instructions ifE, ifS and ifZ are available.

The:

```
    int  B:  2; /* Bits: 00 = 32, 01 = 16, 10 = 8 */
```

flag is used to control the number of bits moved by the movdi() and movid() instructions. The number of bits to be moved is toggled between the three states 8, 16 and 32 by using the:

```
void tbf() /* toggle bits flag */
```

instruction. The:

```
    int  D:  1; /* Direction: for shift, rotate, move, search */
```

flag is used to control the direction of shift, rotate, move and search instructions. The value of this flag is toggled by the

```
void tdf() /* toggle direction flag */
```

instruction. However, at present, no instructions use this feature.

The other set of flags for the CPU uses the CRflags structure:

```
    CRflags cf;          /* CPU Register flags */

typedef struct {
    TRind  Se; /* current segment register */
    TRind  De; /* current destination register */
    TRind  So; /* current  source register */
} CRflags; /* CPU register flags */

typedef struct {
    I8s  i; /* index into Cpu.re array of registers */
    I8s  t; /* index into Rtog.r list of registers */
} TRind; /* register toggle indexes */
    /* note that TRind.t will be toggled by incrementing it with the
       modulus Rtog.n, after each toggle, TRind.i will be updated to
       provide a more direct reference to the actual Cpu.re register */
```

When the GetAMap function reads the opcode.map file during the Tierra startup, it looks for lines at the beginning of the opcode.map file that start with Se, De or So. If it finds these lines, it reads a list of registers that could be used as the segment, destination and source registers (respectively). These lists of registers are placed into the IDRegs structure:

```
typedef struct {  /* there will be only one global copy of this structure */
    Rtog  Se; /* list of segment registers */
    Rtog  De; /* list of destination registers */
    Rtog  So; /* list of source registers */
} IDRegs;

typedef struct {
    I8s  n;   /* number of register to toggle */
    I8s  *r;  /* list of n registers to toggle */
} Rtog; /* for keeping track of toggled registers */
```

Now any register in the lists can be used as the segment, destination or source registers, by toggling through the list using the instructions:

```
void ter() /* toggle segment registers */
void tdr() /* toggle destination registers */
void tsr() /* toggle source registers */
```

These instructions toggle the values in the CRflags structure. See the ter(), tdr(), and tsr() instructions in the instruct.c file. The toggled segment, source and destination registers are used by instruction like movdi(). Here is some relevant code for the movdi() instruction from the decode.c file:

```
        if (ce->c.d->idf.De) /* using destination toggle registers */
            tval0 = flaw() + ce->c.c->cf.De.i;
        else /* using fixed destination registers provided by opcode.map */
            tval0 = flaw() + ce->c.d->re[0];
        if (ce->c.d->idf.So) /* using source toggle registers */
```

```
            tval1 = flaw() + ce->c.c->cf.So.i;
        else /* using fixed source registers provided by opcode.map */
            tval1 = flaw() + ce->c.d->re[1];
        if (ce->c.d->idf.Se) /* using segment toggle registers */
            tval2 = flaw() + ce->c.c->cf.Se.i;
        else
            tval2 = flaw() + ce->c.d->re[2];
```

Note that the segment, source, and destination registers are used conditional on the corresponding flags being set in the:

```
    IDflags  idf;              /* flags usage */
```

element of the InstDef structure.

8.1) Run-Time Configuration of Instruction Sets

One of the major innovations to first appear in version 4.1 is the ability to define most aspects of the instruction set at run time, rather than at compile time. Before version 4.1, the four sets 0, 1, 2, and 3 had to be defined at compile time. Now it is possible to configure the system for any of these four sets, and many more, at run-time. In some sense this innovation makes the instruction sets disappear as well defined entities, because it is so easy to alter them. Even so, the first four instruction sets have been preserved by creating opcode.map files (see below) that preserve their original characteristics.

The run-time configuration of the instruction set is determined by the opcode.map file that is pointed to by the IMapFile variable of the soup_in file (si0, si1...). At startup time, the relevant opcode.map is read from disk, by the GetAMap() function contained in the genio.c source module, and its contents are placed into an array of InstDef structures:

```
typedef struct {
    unsigned  Se: 1; /* oFfset - segment */
    unsigned  B:  1; /* Bits */
    unsigned  De: 1; /* Target - destination */
    unsigned  So: 1; /* Get - source */
    unsigned  D:  1; /* Direction */
    unsigned  H:  1; /* sHadow registers */
    unsigned  P:  1; /* reverse Polish */
    unsigned  C:  1; /* speCial */
} IDflags;

typedef struct { /* structure for instruction set definitions */
    I8s op;                      /* op code */
    I8s mn[9];                   /* assembler mnemonic */
    void (*execute) P_((void)); /* pointer to execute function */
    void (*decode) P_((void));  /* pointer to decode function */
    I8s  re[8];                  /* register assignments */
    IDflags  idf;                /* flags usage */
} InstDef;

InstDef *id;
```

8.1.1) The InstDef Structure

Let me explain the elements of this structure:

```
    I8s op;                         /* opcode */
```

This is the opcode, the numeric value from the soup that will be decoded into the executable instruction. In the present configuration of Tierra, this may take on values in the range of 0 to 255.

```
    I8s mn[9];                    /* assembler mnemonic */
```

This is the assembler mnemonic associated with the instruction. The assembler, arg, reads ASCII files and assembles them into executable binary files. Arg reads the assembler mnemonics from the source file and replaces them with the appropriate opcode in the binary file.

```
    void (*execute) P_((Cell  *)); /* pointer to execute function */
```

This is a pointer to the function which will actually carry out the operation associated with this instruction. These functions are defined in the instruct.c source module. Because the InstDef array contains pointers, these can not be specified in the opcode.map file. Therefore the GetAMap() function ignores this field in the opcode.map file.

```
    void (*decode)  P_((Cell  *)); /* pointer to decode function */
```

This is a pointer to the function which will decode the operation associated with this instruction. These functions are defined in the decode.c source module. Because the InstDef array contains pointers, these can not be specified in the opcode.map file. Therefore the GetAMap() function ignores this field in the opcode.map file.

```
    I8s  re[8]; /* register assignments */
```

Many of the executable instructions operate on source values from registers and/or generate destination values that will be placed in registers. For example, add and subtract take two source values and generate one destination value. This re[8] array specifies the registers that will be associated with these source and destination values. The opcode.map file contains a string in this field, in which the letters of the alphabet specify the registers, where the letter "a" corresponds to the first register, "b" to the second register, etc. For example, in set 0, the instruction subAAC uses the following line in the opcode0.map file:

```
    {0, "subAAC", math, dec1d2s, "aac", ""}, /* "aac" */
```

The register assignment field is "aac". The first "a" specifies the destination register, and the next two letters "ac" specify the two source registers. Note the use of ce->c.p->re[#] in the decode.c file. See below for more details.

```
    IDflags  idf;              /* flags usage */
```

This field contains the flags bitfield, whose elements are described below. The flags bitfield is specified by the last string field of the opcode.map file:

```
    {0, "movdd", movdd, dec1d1s, "ab", "H"},   /* "rr" */
```

In this example, the "H" is interpreted as a flag which specifies that this instruction will use shadow registers. The available flags are listed below, from the IDflags structure:

```
    unsigned  Se: 1; /* oFfset - segment */
    unsigned  B:  1; /* Bits */
    unsigned  De: 1; /* Target - destination */
    unsigned  So: 1; /* Get - source */
    unsigned  D:  1; /* Direction */
    unsigned  H:  1; /* sHadow registers */
    unsigned  P:  1; /* reverse Polish */
    unsigned  C:  1; /* speCial */
```

I will explain the usage of each of the above flags:

```
    unsigned  Se: 1; /* oFfset - segment */
```

Used to indicate the we are using the option of toggling the segment register. Look for code triggered by if(ce->c.d->idf.Se) in the decode.c file.

```
    unsigned  B:  1; /* Bits */
```

When moving data between the CPU registers and the soup, we can move 8 bits, 16 bits, or 32 bits. If this flag is set, a mov instruction can move different numbers of bits based on the value of the ce->c.c->fl.B flag. The value of this flag can be toggled between the three states using the tbf() instruction in instruct.c. This is implemented by the following code from the decode.c file:

```
    switch (ce->c.d->mn[5])
    {   case 0: /* movdi() */
        {   if (ce->c.d->idf.B)
            {   switch (ce->c.c->fl.B)
                {   case 0: goto bit32;
                    case 1: goto bit16;
                    case 2: goto bit8;
                }
            }
            is.sval2 = *(is.dins);
            break;
        }
        case '8': /* movdi8() */
        {   bit8:
            i8 = (I8s *) is.dins;
            is.sval2 = *(i8);
            break;
        }
        case '1': /* movdi16() */
        {   bit16:
            i16 = (I16s *) is.dins;
            is.sval2 = *(i16);
            break;
        }
        case '3': /* movdi32() */
        {   bit32:
            i32 = (I32s *) is.dins;
            is.sval2 = *(i32);
            break;
        }
    }

    unsigned  De: 1; /* Target - destination */
```

Used to indicate the we are using the option of toggling the destination register. Look for code triggered by if(ce->c.d->idf.De) in the decode.c file.

```
    unsigned  So: 1; /* Get - source */
```

Used to indicate the we are using the option of toggling the source register. Look for code triggered by if(ce->c.d->idf.So) in the decode.c file.

```
    unsigned  D:  1; /* Direction */
```

Currently unused, but available for controlling the directions of operations such as shift, rotate, move, search, etc.

```
    unsigned  H:  1; /* sHadow registers */
```

Instruction set 1 uses an indirect method of referencing the registers, through shadow registers, so that any instruction can operate on any register(s). This shadow referencing is now available in any instrucion set just by placing "H" in the flags string field of the opcode.map file. Note that the use of shadow referencing is done on an instruction by instruction basis, so that any actual instruction set can mix shadow referencing with direct referencing. Note the use of the "H" character in the opcode1.map file, and note the code triggered by if(ce->c.d->idf.H) in the decode.c file.

```
    unsigned  P:  1; /* reverse Polish */
```

Instruction set 2 uses a Reverse Polish Notation, which requires that registers be shifted up or down after some instructions. This shifting of registers is now available in any instrucion set just by placing "P" in the flags string field of the opcode.map file. Note that the use of register shifting is done on an instruction by instruction basis, so that any actual instruction set can mix register shifting with normal register access. Note the use of the "P" in the opcode2.map file, and note the code triggered by if(ce->c.d->idf.P) in the instruct.c file.

```
    unsigned  C:  1; /* speCial */
```

There are some situations in which the relationship of the source and destination values to registers in not simple enough to be handled by the mechanisms implemented through the re[8] field. In these cases a flag is set, to indicate that special handling is necessary. Flag conditions are triggered by the presence of characters in the register assignments field which are not lower case letters. For example, in instruction set 3, the movii() instruction moves information to and from addresses specified by the ax and bx registers, but these address are actually offsets from the address specified by the cx register. This relatively complex condition is triggered by the flag in the movii() instruction. Note the code triggered by if(ce->c.d->idf.C) in pmovii() and other functions in the decode.c file.

8.1.2) The GetAMap Function

The run-time configuration of the instruction set is determined by the opcode.map file that is pointed to by the IMapFile variable of the soup_in file (si0, si1...). At startup time, the relevant opcode.map is read from disk, by the GetAMap() function contained in the genio.c source module, and its contents are placed into an array of InstDef structures named "id".

At initialization, GetAMap assumes a one bit instruction set (with two instructions), and allocates id as an array of two InstDef structures. Then GetAMap begins reading the opcode.map file. If the opcode.map file defines more than two instructions, when GetAMap reads the third instruction from the opcode.map file, it will assume a two bit instruction set, and reallocate id as an array of four InstDef structures. Each time that the number of instructions specified by the opcode.map file exceeds the size of the id array of InstDef structures, GetAMap assumes one more bit in the instruction set and doubles the size of the id array. If the opcode.map file specifies a number of instructions that is not a power of two, GetAMap will pad out the instruction set by mirroring the lower half of the instruction set into the unfilled portion of the upper half.

The opcode.map files included with the distribution are configured to reflect the InstDef structure. However, the GetAMap function only reads the three quote delimited string fields:

```
    {0, "movdd", movdd, dec1d1s, "ab", "H"},   /* "rr" */
```

The contents of the first string are written into mn[9]. The contents of the second field are written into re[8], and the third field is decoded into the

```
IDflags  idf;              /* flags usage */
```

field. Because GetAMap recognizes these three fields by their enclosure in quotes, the opcode.map files need only include these three fields, and they must be delimited by quotes. For example:

```
    "movdd" "ab" "H"
```

GetAMap will ignore lines that do not scan properly for these three fields.

The opcodes are assigned to the instructions in the order that they appear in opcode.map file, starting with 0x00 and counting up in increments of one.

The soup_in.h file defines an array of InstDef structures named idt, which contains the superset of available instructions. The only fields of this structure that are used are:

```
    I8s mn[9];                     /* assembler mnemonic */
    void (*execute) P_((Cell  *)); /* pointer to execute function */
    void (*decode)  P_((Cell  *)); /* pointer to decode function */
```

Each time that GetAMap reads a line from the opcode.map file, it compares the assembler mnemonic field to the mnemonics listed in the idt array. If it finds a matching mnemonic, it copies the relevant structure over from the idt array to the current location in the id array. It then assigns an opcode to the current element of the id array, based on a simple count of the number of instructions processed.

Then GetAMap processes the register assignments field. It does this by translating the letters, a, b, c, d... into the numeric values, 0, 1, 2, 3... The numbers are used to reference the relevant register from the array of registers in the Cpu structure. Capital letters will also be translated into numbers, but they will cause the flag field to be set in the InstDef structure. Similarly, actual digits will be mapped into their numeric values, and they will cause the flag field to be set. If a space is encountered in the register assignment field, then its corresponding position in the re[] field of the InstDef structure will be assigned a value of -1 and the flag field will be set.

When the "H" character is encountered in the third string field, the sHadow field is set. When the "P" character is encountered in the third string field, the Polish the rpn (Polish) field is set.

Where more than one register is referenced, the relationship between the order of the registers listed in the register assignment field, and their actual functional roles is determined by the decode functions. Generally, the destination registers preceed the source registers, but the listing of register references in the opcode.map files must be done carefully on a case by case basis, by studying the corresponding decode function that will process the registers.

The flag conditions are processed on a case by case basis in the decode functions. This is not elegant, but for now it works.

Each time an instruction is fetched by the FetchDecode() function, a pointer in the Cpu structure, ce->c.p, is set to point to the InstDef structure from the id array corresponding to the instruction just fetched. This InstDef structure is then used in the decoding that follows the fetch.

8.2) Available instructions, the InstDef idt[] array

The idt array of InstDef structures defines the set of instructions available for inclusion in the instruction set at run-time. The sub-set of these actually used is determined by the contents of the opcode.map file. The current contents of the idt array is listed below:

```
InstDef idt[] = {
    {0, "nop0", nop, pnop, "", 0},              /* no decode args */
    {0, "nop1", nop, pnop, "", 0},              /* no decode args */
    {0, "add", add, dec1d2s, "", 0},            /* "rrr" */
    {0, "add2", add, dec1d2s, "", 0},           /* "rrr" */
    {0, "adrb", adr, decadr, "", 0},            /* "rr  " */
    {0, "adrf", adr, decadr, "", 0},            /* "rr  " */
    {0, "adro", adr, decadr, "", 0},            /* "rr  " */
    {0, "and", and, dec1d2s, "", 0},            /* "rrr" */
    {0, "and2", and, dec1d2s, "", 0},           /* "rrr" */
    {0, "call", tcall, ptcall, "", 0},          /* no decode args */
    {0, "clrf", clrf, pnop, "", 0},             /* no decode args */
    {0, "clrfi", clrfi, pnop, "", 0},           /* no decode args */
    {0, "clrrf", clrrf, pnop, "", 0},           /* no decode args */
    {0, "csync", csync, pnop, "", 0},           /* no decode args */
    {0, "dec", add, dec1d1s, "", 0},            /* "rr" */
    {0, "decC", add, dec1d1s, "cc", 0},         /* "cc" */
    {0, "div", idiv, dec1d2s, "", 0},           /* "r" */
    {0, "div2", idiv, dec1d2s, "", 0},          /* "r" */
    {0, "divide", divide, dec2s, "", 0},        /* "rr" */
    {0, "enter", enter, pnop, "", 0},           /* no decode args */
    {0, "exch", exch, dec2s, "", 0},            /* "rr" */
    {0, "getregs", getregs, dec1s, "t", 0},     /* "#" */
    {0, "ifE", skip, dec2s, "", 0},             /* "rr" */
    {0, "ifS", skip, dec2s, "", 0},             /* "rr" */
    {0, "ifZ", skip, dec2s, "", 0},             /* "rr" */
    {0, "ifequal", skip, dec2s, "", 0},         /* "rr" */
    {0, "ifgrtr", skip, dec2s, "", 0},          /* "rr" */
    {0, "ifless", skip, dec2s, "", 0},          /* "rr" */
    {0, "ifz", skip, dec2s, "", 0},             /* "rr" */
    {0, "inc", add, dec1d1s, "", 0},            /* "rr" */
    {0, "incA", add, dec1d1s, "aa", 0},         /* "aa" */
    {0, "incB", add, dec1d1s, "bb", 0},         /* "bb" */
    {0, "incC", add, dec1d1s, "cc", 0},         /* "cc" */
    {0, "jmpb", adr, decjmp, "", 0},            /* "r" */
    {0, "jmpf", adr, decjmp, "", 0},            /* "r" */
    {0, "jmpo", adr, decjmp, "", 0},            /* "r" */
    {0, "join", join, pnop, "", 0},             /* no decode args */
    {0, "mal", malchm, dec1d3s, "", 0},         /* "rr r" */
    {0, "movBA", movdd, dec1d1s, "ba", 0},      /* "ba" */
    {0, "movDC", movdd, dec1d1s, "dc", 0},      /* "dc" */
    {0, "movdd", movdd, dec1d1s, "", 0},        /* "rr" */
    {0, "movdi", movdi, pmovdi, "", 0},         /* "rr" */
    {0, "movdi8", movdi, pmovdi, "", 0},        /* "rr" */
    {0, "movdi16", movdi, pmovdi, "", 0},       /* "rr" */
    {0, "movdi32", movdi, pmovdi, "", 0},       /* "rr" */
    {0, "movid", movid, pmovid, "", 0},         /* "rr" */
    {0, "movid8", movid, pmovid, "", 0},        /* "rr" */
    {0, "movid16", movid, pmovid, "", 0},       /* "rr" */
    {0, "movid32", movid, pmovid, "", 0},       /* "rr" */
    {0, "movii", movii, pmovii, "", 0},         /* "rr" */
    {0, "mul", mul, dec1d2s, "", 0},            /* "r" */
    {0, "mul2", mul, dec1d2s, "", 0},           /* "r" */
    {0, "not", not, dec1d, "", 0},              /* "r" */
    {0, "not0", not0, dec1d, "", 0},            /* "r" */
    {0, "offAACD", offset, dec1d3s, "aacd", 0}, /* "aacd" */
    {0, "offBBCD", offset, dec1d3s, "bbcd", 0}, /* "bbcd" */
    {0, "offset", offset, dec1d3s, "", 0},      /* "rrrr" */
    {0, "ior", ior, dec1d2s, "", 0},            /* "rrr" */
    {0, "ior2", ior, dec1d2s, "", 0},           /* "rrr" */
    {0, "pop", pop, dec1d, "", 0},              /* "r" */
    {0, "popA", pop, dec1d, "a", 0},            /* "a" */
    {0, "popB", pop, dec1d, "b", 0},            /* "b" */
    {0, "popC", pop, dec1d, "c", 0},            /* "c" */
    {0, "popD", pop, dec1d, "d", 0},            /* "d" */
    {0, "popE", pop, dec1d, "e", 0},            /* "e" */
    {0, "popF", pop, dec1d, "f", 0},            /* "f" */
    {0, "push", push, dec1s, "", 0},            /* "r" */
    {0, "pushA", push, dec1s, "a", 0},          /* "a" */
    {0, "pushB", push, dec1s, "b", 0},          /* "b" */
    {0, "pushC", push, dec1s, "c", 0},          /* "c" */
    {0, "pushD", push, dec1s, "d", 0},          /* "d" */
    {0, "pushE", push, dec1s, "e", 0},          /* "e" */
    {0, "pushF", push, dec1s, "f", 0},          /* "f" */
    {0, "rand", movdd, dec1d1s, "", 0},         /* "rr" */
    {0, "ret", pop, dec1d, "", 0},              /* no decode args */
    {0, "rolld", rolld, pnop, "", 0},           /* no decode args */
    {0, "rollu", rollu, pnop, "", 0},           /* no decode args */
    {0, "shl", shl, dec1d, "", 0},              /* "r" */
    {0, "shr", shr, dec1d, "", 0},              /* "r" */
    {0, "split", split, dec1d1s, "", 0},        /* "rr" */
    {0, "sub", add, dec1d2s, "", 0},            /* "rrr" */
    {0, "sub2", add, dec1d2s, "", 0},           /* "rrr" */
    {0, "subCAB", add, dec1d2s, "cab", 0},      /* "cab" */
    {0, "subAAC", add, dec1d2s, "aac", 0},      /* "aac" */
    {0, "surf", migrate, dec1s, "", 0},         /* "r" */
    {0, "tbf", tbf, pnop, "", 0},               /* no decode args */
    {0, "tdf", tdf, pnop, "", 0},               /* no decode args */
    {0, "ter", ter, pnop, "", 0},               /* no decode args */
    {0, "tsr", tsr, pnop, "", 0},               /* no decode args */
    {0, "tdr", tdr, pnop, "", 0},               /* no decode args */
    {0, "xor", xor, dec1d2s, "", 0},            /* "rrr" */
    {0, "xor2", xor, dec1d2s, "", 0},           /* "rrr" */
    {0, "zero", movdd, dec1d1s, "", 0},         /* "rr" */
    {0, "zeroD", movdd, dec1d1s, "dd", 0},      /* "dd" */
#ifdef IO
    {0, "get", get, dec1d1s, "", 0},            /* "r" */
    {0, "put", put, dec1d1s, "", 0},            /* "rr" */
    {0, "puticc", puticc, pputicc, "", 0},      /* "rr" */
#endif /* IO */
#if PLOIDY > 1
    {0, "trso", trso, pnop, "", 0},             /* no decode args */
    {0, "trde", trde, pnop, "", 0},             /* no decode args */
    {0, "trex", trex, pnop, "", 0},             /* no decode args */
#else /* PLOIDY > 1 */
    {0, "trso", NULL, NULL, "", 0},
    {0, "trde", NULL, NULL, "", 0},
    {0, "trex", NULL, NULL, "", 0},
#endif /* PLOIDY > 1 */
#ifdef SHADOW
    {0, "A", regorder, dec1s, "a", 0},          /* "a" */
    {0, "B", regorder, dec1s, "b", 0},          /* "b" */
    {0, "C", regorder, dec1s, "c", 0},          /* "c" */
    {0, "D", regorder, dec1s, "d", 0},          /* "d" */
#ELse /* SHADOW */
    {0, "A", NULL, NULL, "", 0},
    {0, "B", NULL, NULL, "", 0},
    {0, "C", NULL, NULL, "", 0},
    {0, "D", NULL, NULL, "", 0},
#endif /* SHADOW */
#ifdef NET
    {0, "tpings", tpingsnd, dec1s, "", 0},    /* "r" */
    {0, "tpingr", tpingrec, dec1d, "", 0},    /* "r" */
    {0, "getip",  getip,    dec1d, "", 0},    /* "r" */
NEEDED: {0, "getipp",  getipp,    dec1d, "", 0},    /* "r" */
NEEDED: network intercellular communication mechanisms
#endif /* NET */
    {0, "END", NULL, NULL, "", 0}             /* this line must be last! */
};
```

I will now discuss the function and use of each of these instructions in turn. There are some operations that are performed in many of the instructions. Rather than repeating the complete description of these with each instruction, I will describe them first, and then only refer to them by name in the relevant instructions.

```
void DoRPNd()
{   I16s  i;

    if (ce->c.d->idf.P)
        for (i = 1; i < NUMREG - 1; i++)
            ce->c.c->re[i] = ce->c.c->re[i + 1] + flaw();
}
```

```
void DoRPNu()
{   I16s  i;

    if (ce->c.d->idf.P)
        for (i = NUMREG - 1; i > 0; i--)
            ce->c.c->re[i] = ce->c.c->re[i - 1] + flaw();
}
```

Some of the instructions can use a Reverse Polish Notation type of stack, modeled after the Hewlett-Packard calculator. In these instructions, when a math operation is performed, the top three registers descend, using the DoRPNd() function. When data enters the lower register, the registers are moved up using the DoRPNu() function (see above). We will indicate when these functions are performed by stating: DoRPNd or DoRPNu. However, these actions are only performed if the P flag is set: if (ce->c.d->idf.P) The rolld and rollu instruction are used for rolling the RPN stack, and the exch instruction is used for exchanging the values in the lower two registers of the stack.

```
void DoFlags()
{   ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z = 0;
    if (*(is.dreg) == 0)
        ce->c.c->fl.Z = 1;
    if (*(is.dreg) < 0)
        ce->c.c->fl.S = 1;
}
```

After the execution of each instruction, the E, S and Z flags are cleared to zero. However, if a zero value is created in the destination register, the Z flag will be set to one. If a negative value is created in the destination register, the S flag will be set to one. And if some error condition is generated, the instruction will generally perform no operation, while setting the E flag to one. We will indicate that these operations are performed by stating: DoFlags. The ifE, ifS, and ifZ instructions will skip the instruction that follows if the corresponding flag is set.

```
void DoMods()
{   if (is.dmod)
    {   *(is.dreg) = mo(*(is.dreg), is.dmod);
        is.dmod = 0;
    }
    else if (is.dran && (*(is.dreg) > is.dran || *(is.dreg) < -is.dran))
    {   *(is.dreg) = 0;
        is.dran = 0;
    }
}
```

If the decode function sets the value of is.dmod to a non-zero value, then the value in the destination register is modulused (using mo()) to remain in the range of 0 to is.dmod - 1. If the decode function sets the value of is.dran to a non-zero value, then the value in the destination register must fit withing the range -is.dran to is.dran. If the value is outside that range, it is reset ot zero. The presence of these operations in an instruction will be indicated by stating: DoMods

```
void DoMods2()
{   if (is.dmod2)
    {   *(is.dreg2) = mo(*(is.dreg2), is.dmod2);
        is.dmod2 = 0;
    }
    else if (is.dran2 && (*(is.dreg2) > is.dran2 || *(is.dreg2) < -is.dran2))
    {   *(is.dreg2) = 0;
        is.dran2 = 0;
    }
}

void DoMods3()
{   if (is.dmod3)
```

```
    {   *(is.dreg3) = mo(*(is.dreg3), is.dmod3);
        is.dmod3 = 0;
    }
    else if (is.dran3 && (*(is.dreg3) > is.dran3 || *(is.dreg3) < -is.dran3))
    {   *(is.dreg3) = 0;
        is.dran3 = 0;
    }
}
```

The DoMods function can be performed on the value in a second or third destination register by having the decode function set non-zero values for is.dmod2, is.dreg2, is.dmod3, or is.dran3. The presence of these functions are indicated by stating: DoMods2 or DoMods3

The decode functions set values in the global PInst structure is. The PInst structure has the following form:

```
typedef struct { /* struct for passing arguments from decode to execute */

    Preg   sreg; /* pointer to source register */
    FpIns  sins; /* pointer to source instruction */
    I32s   sval; /* source value */
    I8s    stra; /* source track */

    Preg   dreg;  /* pointer to destination register */
    I32s   dval;  /* original destination value */
    FpIns  dins;  /* pointer to destination instruction */
    I8s    dtra;  /* destination track */
    I32s   dmod;  /* destination modulused positive this size */
    I32s   dran;  /* destination kept in signed range of this value */
    Pcells dcel;  /* destination cell */

    Preg   sreg2; /* pointer to 2nd source register */
    FpIns  sins2; /* pointer to 2nd source instruction */
    I32s   sval2; /* 2nd source value */
    I8s    stra2; /* 2nd source track */

    Preg   dreg2; /* pointer to 2nd destination register */
    I32s   dval2; /* original destination value */
    FpIns  dins2; /* pointer to 2nd destination instruction */
    I8s    dtra2; /* 2nd destination track */
    I32s   dmod2; /* 2nd dest modulused positive this size */
    I32s   dran2; /* 2nd dest kept in signed range of this value */

    I32s   sval3; /* 3rd source value */
    I32s   dval3; /* original destination value */

    Preg   dreg3; /* pointer to 3rd destination register */
    I32s   dmod3; /* 3rd dest modulused positive this size */
    I32s   dran3; /* 3rd dest kept in signed range of this value */

    I8s    mode;  /* mode of instruction */
    I8s    mode2; /* 2nd mode of instruction */
    I8s    mode3; /* 3rd mode of instruction */

    I8s    expr;  /* execute protection 0 = no protection, 1 = protected */
    I32s   oip;   /* address of instruction being executed: ce->c.ip */
    FpIns  eins;  /* pointer to instruction being executed */
    I8s    iip;   /* amount to increment instruction pointer */
    I8s    dib;   /* amount to decrement instruction bank */
    I16s   ts;    /* size of time slice, used to control central loop */
} PInst;
```

In general the decode function will place the source values in the sval, sval2, sval3 elements of is, and it will place a pointer to the destination register in the dreg, dreg2 etc. elements of is. This structure is used to pass data between the decode and the execute functions. The usage of the elements of is will be stated with the description of each instruction.

```
    {0, "nop0", nop, pnop, "", 0},              /* no decode args */
    {0, "nop1", nop, pnop, "", 0},              /* no decode args */
```

These two instructions are no-ops, they do nothing, other than to increment the instruction pointer one place. However, they are used to form the templates for pattern matching instructions like adrb, adrf, adro, call, jmpb, jmpf, and jmpo. There are no conditions under which the nop instructions fail.

```
    {0, "add", add, dec1d2s, "", 0},            /* "rrr" */

 * void add() *(is.dreg) = is.sval + is.sval2;
 * is.dreg  = destination register, where calculation will be stored
 * is.sval  = a value that will be added to is.sval2 and placed in dest reg
 * is.sval2 = a value that will be added to is.sval  and placed in dest reg
 * is.dmod  = value by which to modulus destination register
 * is.dran  = range within which to contain destination register
```

The add instruction takes two source values, adds them, and puts the result in a destination register. The source and destination registers are specified in the fifth field of InstDef (re). The flaw is added to the first source value by the decode function, dec1d2s(). There are no conditions under which the add instruction will fail. DoMods, DoFlags, DoRPNd.

```
    {0, "add2", add, dec1d2s, "", 0},           /* "rrr" */
```

add2 differs from add only in the decode stage. This instruction is used to handle the tricky case that the De or So flags are set, and the destination register also holds one of the source values. In essence this form of the instruction uses two rather than three registers, and one or both of them are specified by the De and So flags.

```
    {0, "adrb", adr, decadr, "", 0},            /* "rr  " */

 * void adr() find address of a template
 * is.mode  = search mode: 1 = forward, 2 = backward, 0 = outward
 * is.mode2 =  preference: 1 = forward, 2 = backward, and return for
 *        direction found: 1 = forward, 2 = backward, 3 = both, 0 = none
 * is.dval  = starting address for forward search
 *               and return for finish address
 * is.dval2 = starting address for backward search
 *               and return for finish address
 * is.dreg  = destination register where target address will be stored
 * is.dreg2 = destination register where template size will be stored
 * is.dreg3 = destination register where offset of target will be stored
 * is.sval  = return address if template size = 0
 * is.sval2 = template size, 0 = no template
 * is.sval3 = search limit, and return for distance actually searched
 * is.dmod  = modulus value for is.dreg
 * is.dmod2 = modulus value for is.dreg2
 * is.dmod3 = modulus value for is.dreg3
 * is.dran  = range to maintain for is.dreg
 * is.dran2 = range to maintain for is.dreg2
 * is.dran3 = range to maintain for is.dreg3
```

This instruction causes the first destination register to contain the address of the instruction following the nearest occurrence (backwards in the soup) of the template complementary to the template that follows the adrb instruction. The template is defined as the group of consecutive nops (nop0 or nop1) that immediately follow the adrb instruction. The size of the template is just the number of consecutive nops that follow the adrb instruction. The size of the template is placed in the second destination register. The target template can be larger than the source template (the entire source template must be matched, but it may match to a sub-template in the target template).

This instruction can use up to four registers, which must be specified in the fifth field (re) of the InstDef structure:

1) destination register for the target template address 2) destination register for the template size 3) destination register for the offset of the target template from the source template 4) source register containing offset from source template, from which to start the search for the target template

Because this instruction could use four registers, it is also optional to not use them all. By setting the C flag, we can not use the registers that are not specified in the re field of InstDef. So for example:

```
    {0, "divide", divide, dec2s, "ac", "C"}
```

The re field specifies only two registers: a, c, so the a register receives the address of the target template, and the c register receives the size of the template. The C flag in the sixth field, causes the unspecified registers to be directed to the bit bucket.

This instruction is a uni-directional search, which means that it will search in only one direction (in this case, only backward) in memory for the complementary template. Let a equal the address of the first instruction of the template, and let s equal the size of the template. The backward search starts at the address a - s - 1. The search looks in successive steps backwards until the complementary template is found, or the search limit (Search_limit) is reached. The template searches wrap around the ends of the soup. The instruction pointer is moved past the template used by the adrb instruction: ce->c.ip = ce->c.ip + s + 1.

The instruction will fail if the size of the template following the adrb instruction is less than MinTemplSize, or greater than SoupSize, or if a complementary template is not found within the search radius, Search_limit. On failure the cx register is not altered. DoMods, DoMods2, DoMods3.

```
    {0, "adrf", adr, decadr, "", 0},            /* "rr  " */
    {0, "adro", adr, decadr, "", 0},            /* "rr  " */
```

These instructions operate like adrb, but adrf searches forward, while adro searches both forward and backward. The forward search starts at the address a + s + 1. DoMods, DoMods2, DoMods3.

```
    {0, "and", and, dec1d2s, "", 0},            /* "rrr" */

 * void and() *(is.dreg) = is.sval & is.sval2;
 * is.dreg  = destination register, where calculation will be stored
 * is.sval  = value that will be anded by is.sval2 and placed in dest reg
 * is.sval2 = value that will be anded into is.sval and placed in dest reg
 * is.dmod  = value by which to modulus destination register
 * is.dran  = range within which to contain destination register
```

The AND instruction takes two source values, performs a bitwise AND operation on them, and puts the result in a destination register. The source and destination registers are specified in the fifth field of InstDef (re). The flaw is added to the first source value by the decode function, dec1d2s(). This instruction will fail if the second source value is zero. DoMods, DoFlags, DoRPNd.

```
    {0, "and2", and, dec1d2s, "", 0},           /* "rrr" */
```

and2 differs from and only in the decode stage. This instruction is used to handle the tricky case that the De or So flags are set, and the destination register also holds one of the source values. In essence this form of the instruction uses two rather than three registers, and one or both of them are specified by the De and So flags.

```
    {0, "call", tcall, ptcall, "", 0},          /* no decode args */

 * void adr() find address of a template
 * is.mode  = search mode: 1 = forward, 2 = backward, 0 = outward
 * is.mode2 =  preference: 1 = forward, 2 = backward, and return for
 *       direction found: 1 = forward, 2 = backward, 3 = both, 0 = none
 * is.dval  = starting address for forward search
 * is.dval2 = starting address for backward search
 * is.dreg  = destination register where target address will be stored
 * is.dreg2 = destination register where template size will be stored
 * is.dreg3 = destination register where offset of target will be stored
 * is.sval  = return address if template size = 0
 * is.sval2 = template size, 0 = no template
 * is.sval3 = search limit, and return for distance actually searched
 * is.dmod  = modulus value for is.dreg
 * is.dmod2 = modulus value for is.dreg2
 * is.dmod3 = modulus value for is.dreg3
 * is.dran  = range to maintain for is.dreg
 * is.dran2 = range to maintain for is.dreg2
 * is.dran3 = range to maintain for is.dreg3
 *
 * void push()
 * is.sval = value to be pushed onto the stack
```

This instruction performs a call through template matching. The IP is pushed onto the stack, and then is jumped to the instruction following the complementary template. This operation is performed by combining adr() with push(). Thus all the details of the adro instruction apply here.

```
    {0, "clrf", clrf, pnop, "", 0},             /* no decode args */
```

This instruction clears all the Cpu.Flags:

```
void clrf() /* clear all Cpu.Flags */
{   ce->c.c->fl.E = ce->c.c->fl.S  = ce->c.c->fl.Z =
    ce->c.c->fl.B = ce->c.c->fl.D = 0;
}
```

```
    {0, "clrfi", clrfi, pnop, "", 0},             /* no decode args */
```

This instruction clears all the Cpu.Flags associated with the CPU cycle:

```
void clrfi() /* clear Cpu.Flags associated with CPU cycles */
{   ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z = 0;
}
```

```
    {0, "clrrf", clrrf, pnop, "", 0},           /* no decode args */
```

This instruction clears all the Cpu.CRflags:

```
void clrrf() /* clear all Cpu.CRflags */
{   ce->c.c->cf.Se.t = 0;
    ce->c.c->cf.Se.i = IDregs.Se.r[0];
    ce->c.c->cf.De.t = 0;
    ce->c.c->cf.De.i = IDregs.De.r[0];
    ce->c.c->cf.So.t = 0;
    ce->c.c->cf.So.i = IDregs.So.r[0];
    ce->c.c->fl.E = ce->c.c->fl.S = ce->c.c->fl.Z = 0;
}
```

```
    {0, "csync", csync, pnop, "", 0},           /* no decode args */
```

This instruction is used to synchronize all the CPUs within a cell. Once a CPU executes this instruction, it stops executing until all the other CPUs within this cell have also executed this instruction. This causes all the CPUs of the cell to start together, at whatever instructions follow their csync instructions.

```
    {0, "dec", add, dec1d1s, "", 0},            /* "rr" */
    {0, "decC", add, dec1d1s, "cc", 0},         /* "cc" */
    {0, "div", idiv, dec1d2s, "", 0},          /* "r" */
    {0, "div2", idiv, dec1d2s, "", 0},         /* "r" */
    {0, "divide", divide, dec2s, "", 0},       /* "rr" */
    {0, "enter", enter, pnop, "", 0},          /* no decode args */
    {0, "exch", exch, dec2s, "", 0},           /* "rr" */

    {0, "getregs", getregs, dec1s, "t", 0},    /* "#" */
```

This causes the CPU executing the instruction, to copy the contents of the registers of another CPU of the same cell, to its own registers. This instruction will fail if the cell has only one CPU. The argument in the re field of the InstDef structure will specify a value 0, 1, or 2, to indicate the mode by with the data will be transferred between CPUs. If the value is zero, the data will be copied from the next lower CPU. If the value is one, the data will be copied from the next higher CPU. If the value is 2, the data will be copied from another CPU chosen at random (but not the same CPU as the one executing the instruction).

```
    {0, "ifE", skip, dec2s, "", 0},            /* "rr" */
    {0, "ifS", skip, dec2s, "", 0},            /* "rr" */
    {0, "ifZ", skip, dec2s, "", 0},            /* "rr" */
    {0, "ifequal", skip, dec2s, "", 0},        /* "rr" */
    {0, "ifgrtr", skip, dec2s, "", 0},         /* "rr" */
    {0, "ifless", skip, dec2s, "", 0},         /* "rr" */
    {0, "ifz", skip, dec2s, "", 0},            /* "rr" */
    {0, "inc", add, dec1d1s, "", 0},           /* "rr" */
    {0, "incA", add, dec1d1s, "aa", 0},        /* "aa" */
    {0, "incB", add, dec1d1s, "bb", 0},        /* "bb" */
    {0, "incC", add, dec1d1s, "cc", 0},        /* "cc" */
    {0, "jmpb", adr, decjmp, "", 0},           /* "r" */
    {0, "jmpf", adr, decjmp, "", 0},           /* "r" */
    {0, "jmpo", adr, decjmp, "", 0},           /* "r" */
```

NEED: halt, to eliminate a cpu, or if there is only one cpu, to terminate the entire cell.

```
    {0, "join", join, pnop, "", 0},            /* no decode args */
```

This instruction causes the CPU executing this instruction to terminate, unless it is the original CPU for the cell. If it is the original CPU, it will wait untill all the other CPUs have executed the join instruction, then it will continue execution. If the CPU executing the join instruction is other than the original CPU, it terminates immediately, and its Cpu structure is deallocated. There is no condition under which this instruction can fail.

```
    {0, "mal", malchm, dec1d3s, "", 0},        /* "rr r" */
    {0, "movBA", movdd, dec1d1s, "ba", 0},     /* "ba" */
    {0, "movDC", movdd, dec1d1s, "dc", 0},     /* "dc" */
    {0, "movdd", movdd, dec1d1s, "", 0},       /* "rr" */
    {0, "movdi", movdi, pmovdi, "", 0},        /* "rr" */
    {0, "movdi8", movdi, pmovdi, "", 0},       /* "rr" */
    {0, "movdi16", movdi, pmovdi, "", 0},      /* "rr" */
    {0, "movdi32", movdi, pmovdi, "", 0},      /* "rr" */
    {0, "movid", movid, pmovid, "", 0},        /* "rr" */
    {0, "movid8", movid, pmovid, "", 0},       /* "rr" */
    {0, "movid16", movid, pmovid, "", 0},      /* "rr" */
    {0, "movid32", movid, pmovid, "", 0},      /* "rr" */
    {0, "movii", movii, pmovii, "", 0},        /* "rr" */
    {0, "mul", mul, dec1d2s, "", 0},           /* "r" */
    {0, "mul2", mul, dec1d2s, "", 0},          /* "r" */
    {0, "not", not, dec1d, "", 0},             /* "r" */
    {0, "not0", not0, dec1d, "", 0},           /* "r" */
    {0, "offAACD", offset, dec1d3s, "aacd", 0}, /* "aacd" */
    {0, "offBBCD", offset, dec1d3s, "bbcd", 0}, /* "bbcd" */
    {0, "offset", offset, dec1d3s, "", 0},     /* "rrrr" */
    {0, "ior", ior, dec1d2s, "", 0},           /* "rrr" */
    {0, "ior2", ior, dec1d2s, "", 0},          /* "rrr" */
    {0, "pop", pop, dec1d, "", 0},             /* "r" */
    {0, "popA", pop, dec1d, "a", 0},           /* "a" */
    {0, "popB", pop, dec1d, "b", 0},           /* "b" */
    {0, "popC", pop, dec1d, "c", 0},           /* "c" */
    {0, "popD", pop, dec1d, "d", 0},           /* "d" */
    {0, "popE", pop, dec1d, "e", 0},           /* "e" */
    {0, "popF", pop, dec1d, "f", 0},           /* "f" */
    {0, "push", push, dec1s, "", 0},           /* "r" */
    {0, "pushA", push, dec1s, "a", 0},         /* "a" */
    {0, "pushB", push, dec1s, "b", 0},         /* "b" */
    {0, "pushC", push, dec1s, "c", 0},         /* "c" */
    {0, "pushD", push, dec1s, "d", 0},         /* "d" */
    {0, "pushE", push, dec1s, "e", 0},         /* "e" */
    {0, "pushF", push, dec1s, "f", 0},         /* "f" */
    {0, "rand", movdd, dec1d1s, "", 0},        /* "rr" */
    {0, "ret", pop, dec1d, "", 0},             /* no decode args */
    {0, "rolld", rolld, pnop, "", 0},          /* no decode args */
    {0, "rollu", rollu, pnop, "", 0},          /* no decode args */
    {0, "shl", shl, dec1d, "", 0},             /* "r" */
    {0, "shr", shr, dec1d, "", 0},             /* "r" */

    {0, "split", split, dec1d1s, "", 0},       /* "rr" */

 * is.sval = source value for processor number
 * is.dreg = destination register for processor number
 * is.dval = destination register # for new processor number
```

This instruction creates a new CPU for the same cell. The entire contents of the Cpu structure are copied from the mother to the daughter CPU (registers, instruction pointer, stack, stack pointer, flags, etc.). However, one register is altered. The register is specified by the re field of the InstDef structure. In the mother CPU, the value of this register is doubled, while the same register in the daughter CPU gets the doubled value plus one. This procedure provides unique values in all the CPUs that could be generated for a cell, and these unique values could be used to differentiate the functions of the CPUs.

The instruction will fail if generation of a new CPU would cause the total number of CPUs for the cell to be greater than or equal to MaxCpuPerCell, a soup_in variable.

This instruction requires two arguments in the re field of the InstDef structure, and they should be the same. They both refer to the register which will be altered to differentiate the CPUs. One is used to name the register where the altered values will be placed, and the other is used to pass the old value in that register to the execute function (funky but it works).

```
    {0, "sub", add, dec1d2s, "", 0},           /* "rrr" */
    {0, "sub2", add, dec1d2s, "", 0},          /* "rrr" */
    {0, "subCAB", add, dec1d2s, "cab", 0},     /* "cab" */
    {0, "subAAC", add, dec1d2s, "aac", 0},     /* "aac" */

    {0, "surf", migrate, dec1s, "", 0},        /* "r" */

 * is.sval = IP address of the remote node to which creature is to be sent
```

This instruction is used to cause the cell that executes it to be ejected from the soup, and to migrate to the soup of the Tierra running on the machine at the IP address specified in the source register. This causes this cell to be removed from the slicer and reaper queues, and deallocates its memory from the soup. If it has an embryonic daughter, its memory will also be deallocated. However, the mother and daughter's code in the soup is not altered. If the IP address specified in the source register is not valid, it will be mapped to the closest matching (hamming distance) IP in the IP map. There are no conditions under which this instruction can fail. If NET is not defined, then this creature will be removed from the soup, but will not migrate to another node.

```
    {0, "tbf", tbf, pnop, "", 0},              /* no decode args */
    {0, "tdf", tdf, pnop, "", 0},              /* no decode args */
    {0, "ter", ter, pnop, "", 0},               /* no decode args */
    {0, "tsr", tsr, pnop, "", 0},               /* no decode args */
    {0, "tdr", tdr, pnop, "", 0},               /* no decode args */
    {0, "xor", xor, dec1d2s, "", 0},            /* "rrr" */
    {0, "xor2", xor, dec1d2s, "", 0},           /* "rrr" */
    {0, "zero", movdd, dec1d1s, "", 0},         /* "rr" */
    {0, "zeroD", movdd, dec1d1s, "dd", 0},      /* "dd" */

#ifdef IO
    {0, "get", get, dec1d1s, "", 0},            /* "r" */
    {0, "put", put, dec1d1s, "", 0},            /* "rr" */
    {0, "puticc", puticc, pputicc, "", 0},      /* "rr" */
#endif /* IO */

#if PLOIDY > 1
    {0, "trso", trso, pnop, "", 0},             /* no decode args */
    {0, "trde", trde, pnop, "", 0},             /* no decode args */
    {0, "trex", trex, pnop, "", 0},             /* no decode args */
#else /* PLOIDY > 1 */
    {0, "trso", NULL, NULL, "", 0},
    {0, "trde", NULL, NULL, "", 0},
    {0, "trex", NULL, NULL, "", 0},
#endif /* PLOIDY > 1 */

#ifdef SHADOW
    {0, "A", regorder, dec1s, "a", 0},          /* "a" */
    {0, "B", regorder, dec1s, "b", 0},          /* "b" */
    {0, "C", regorder, dec1s, "c", 0},          /* "c" */
    {0, "D", regorder, dec1s, "d", 0},          /* "d" */
#else /* SHADOW */
    {0, "A", NULL, NULL, "", 0},
    {0, "B", NULL, NULL, "", 0},
    {0, "C", NULL, NULL, "", 0},
    {0, "D", NULL, NULL, "", 0},
#endif /* SHADOW */

#ifdef NET /* the following instructions are valid only if NET is defined */

    {0, "tpings", tpingsnd, dec1s, "", 0},    /* "r" */

 * is.sval = IP address of the remote node to which the ping is to be sent
```

This instruction tpings the Tierra process on the machine at the IP address specified in the source register. If the IP address specified in the source register is not valid, it will be mapped to the closest matching (hamming distance) IP in the IP map. There are no conditions under which this instruction can fail.

```
    {0, "tpingr", tpingrec, dec1d, "", 0},    /* "r" */

 * is.dreg = points to location in soup where data will be written
```

This instruction causes a check for a tping response. If one is present, the tping data structure will be written into the soup, at the address specified by the data in the destination register. If there is no tping response available, the instruction will fail and the error flag will be set. If there is a tping response available, it will be written a byte at a time, beginning at the soup address specified by the destination register. If an attempt is made to write to a write protected soup address, the write process will stop, and the error flag will be set.

```
    {0, "getip",  getip,    dec1d, "", 0},    /* "r" */

 * is.dreg = the register where the IP address will be placed
```

Each time this instruction is called, it returns an IP address from the IPMap. Subsequent calls return successive addresses from the list. After returning all the addresses in the list, it returns zero, then resets to the beginning of the list. The address is placed in the destination register. There are no conditions under which this instruction can fail.

NEEDED:

```
    {0, "getipp",  getipp,    dec1d, "", 0},    /* "r" */

 * is.dreg = points to location in soup where data will be written,
            also the register where the IP address will be placed
```

Each time this instruction is called, it writes a tping data structure from the IPMap into the soup, and writes the IP address of that data to the destination register. Subsequent calls return successive structures from the list. After returning all the structures in the list, it returns zero, then resets to the beginning of the list. The IP address is placed in the destination register. There are no conditions under which this instruction can fail.

```
#endif  NET: the preceding instructions are valid only if NET is defined */

    {0, "END", NULL, NULL, "", 0}            /* this line must be last! */
```

8.3) Synopsis of the First Four Sets

Although the concept of well defined instruction sets as immutable entities died with version 4.1, the first four sets have been preserved in the files opcode0.map, opcode1.map, opcode2.map and opcode3.map. Because there has been considerable experience with the use of these four instruction sets, their characteristics are discussed in detail in sections 8.2, 8.3 and 8.4.

```
******************************************************************************

Instset #0  the original instruction set, designed and implemented by
    Tom Ray, in early 1990.

    This instruction set operates on a CPU based on the following definitions:

#define STACK_SIZE 10
#define ALOC_REG 4
#define NUMREG 4         /* NUMREG = ALOC_REG */

typedef struct {         /* structure for registers of virtual CPU */
    Reg re[ALOC_REG];    /* array of registers */
    Reg ip;              /* instruction pointer */
    Reg sp;              /* stack pointer */
    Reg st[STACK_SIZE];  /* stack */
    I8s fl;              /* flag */
} Cpu;

The four registers in the re[] array are referred to as follows:
AX = re[0], BX = re[1], CX = re[2], DX = re[3].

No Operations: 2

nop0
nop1

Memory Movement: 11

pushax (push AX onto stack)
pushbx (push BX onto stack)
pushcx (push CX onto stack)
pushdx (push DX onto stack)
popax  (pop from stack into AX)
popbx  (pop from stack into BX)
```

```
popcx  (pop from stack into CX)
popdx  (pop from stack into DX)
movcd  (DX = CX)
movab  (BX = AX)
movii  (move from ram [BX] to ram [AX])

Calculation: 9

sub_ab (CX = AX - BX)
sub_ac (AX = AX - CX)
inc_a  (increment AX)
inc_b  (increment BX)
inc_c  (increment CX)
dec_c  (decrement CX)
zero   (zero CX)
not0   (flip low order bit of CX)
shl    (shift left all bits of CX)

Instruction Pointer Manipulation: 5

ifz    (if CX == 0 execute next instruction, otherwise, skip it)
jmp    (jump to template)
jmpb   (jump backwards to template)
call   (push IP onto the stack, jump to template)
ret    (pop the stack into the IP)

Biological and Sensory: 5

adr    (search outward  for template, put address in AX, template size in CX)
adrb   (search backward for template, put address in AX, template size in CX)
adrf   (search forward  for template, put address in AX, template size in CX)
mal    (allocate amount of space specified in CX)
divide (cell division)

Total: 32 instructions

The following array defines the associations of opcodes, mnemonics,
executable functions, and decode functions (in that order) used to
implement instruction set #0.  These associations are determined by
the opcode0.map file.

    {0, "nop0", nop, pnop, "", ""},          /* no decode args */
    {0, "nop1", nop, pnop, "", ""},          /* no decode args */
    {0, "not0", not0, dec1d, "c", ""},       /* "r" */
    {0, "shl", shl, dec1d, "c", ""},         /* "r" */
    {0, "zero", movdd, dec1d1s, "cc", ""},   /* "rr" */
    {0, "ifz", skip, dec2s, "cc", ""},       /* "r#" */
    {0, "subCAB", math, dec1d2s, "cab", ""}, /* "cab" */
    {0, "subAAC", math, dec1d2s, "aac", ""}, /* "aac" */
    {0, "incA", math, dec1d1s, "aa", ""},    /* "aa" */
    {0, "incB", math, dec1d1s, "bb", ""},    /* "bb" */
    {0, "decC", math, dec1d1s, "cc", ""},    /* "cc" */
    {0, "incC", math, dec1d1s, "cc", ""},    /* "cc" */
    {0, "pushA", push, dec1s, "a", ""},      /* "a" */
    {0, "pushB", push, dec1s, "b", ""},      /* "b" */
    {0, "pushC", push, dec1s, "c", ""},      /* "c" */
    {0, "pushD", push, dec1s, "d", ""},      /* "d" */
    {0, "popA", pop, dec1d, "a", ""},        /* "a" */
    {0, "popB", pop, dec1d, "b", ""},        /* "b" */
    {0, "popC", pop, dec1d, "c", ""},        /* "c" */
    {0, "popD", pop, dec1d, "d", ""},        /* "d" */
    {0, "jmpo", adr, decjmp, "b", ""},       /* "r" */
    {0, "jmpb", adr, decjmp, "b", ""},       /* "r" */
    {0, "call", tcall, ptcall, "", ""},      /* no decode args */
    {0, "ret", pop, dec1d, "", ""},          /* no decode args */
    {0, "movDC", movdd, dec1d1s, "dc", ""},  /* "dc" */
    {0, "movBA", movdd, dec1d1s, "ba", ""},  /* "ba" */
    {0, "movii", movii, pmovii, "ab", ""},   /* "rr" */
    {0, "adro", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, "adrb", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, "adrf", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, "mal", malchm, dec1d3s, "ac a", ""}, /* "rr r" */
    {0, "divide", divide, dec2s, "aa", "C"},  /* "rr" */

******************************************************************************

Instset #1  Based on a design suggested by Kurt Thearling of Thinking Machines,
    and implemented by Tom Ray.  The novel feature of this instruction set
    is the ability to reorder the relative positions of the registers, using
    the AX, BX, CX and DX instructions.  There are in essence, two sets of
    registers, the first set contains the values that the instruction set
    operates on, the second set points to the first set, in order to determine
    which registers any operation will act on.

    Let the four registers containing values be called AX, BX, CX and DX.
    Let the four registers pointing to these registers be called R0, R1, R2
    and R3.  When a virtual CPU is initialized, R0 points to AX, R1 to BX,
    R2 to CX and R3 to DX.  The instruction "add" does the following:
    (R2 = R1 + R0).  Therefore CX = BX + AX.  However, if we execute the DX
    instruction, the R0 points to DX, R1 to AX, R2 to BX and R3 to CX.  Now
    if we execute the add instruction, we will perform: BX = AX + DX.  If we
    execute the DX instruction again, R0 points to DX, R1 to DX, R2 to AX,
    and R3 to BX.  Now the add instruction would perform: AX = DX + DX.
    Now the registers can be returned to their original configuration by
    executing the following three instructions in order: cx, bx, ax.

    This instruction set operates on a CPU based on the following definitions:

#define SHADOW
#define STACK_SIZE 10
#define ALOC_REG 8
#define NUMREG 4         /* NUMREG = ALOC_REG / 2 */
#define GETBUFSIZ 3
#define PUTBUFSIZ 3

typedef struct {         /* structure for registers of virtual CPU */
    Reg re[ALOC_REG];    /* array of registers */
    Reg ip;              /* instruction pointer */
    Reg sp;              /* stack pointer */
    Reg st[STACK_SIZE];  /* stack */
    I8s fl;              /* flag */
    Reg gb[GETBUFSIZ+3]; /* input buffer */
    Reg pb[PUTBUFSIZ+3]; /* output buffer */
} Cpu;

The eight registers in the re[] array are referred to as follows:
AX = re[0], BX = re[1], CX = re[2], DX = re[3],
R0 = re[4], R1 = re[5], R2 = re[6], R3 = re[7].

No Operations: 2

nop0
nop1

Memory Movement: 12

ax     (make AX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
bx     (make BX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
cx     (make CX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
dx     (make DX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
```

```
movdd  (move R1 to R0)
movdi  (move from R1 to ram [R0])
movid  (move from ram [R1] to R0)
movii  (move from ram [R1] to ram [R0])
push   (push R0 onto stack)
pop    (pop from stack into R0)
put    (write R0 to output buffer, three modes:
         #ifndef ICC: write R0 to own output buffer
         #ifdef ICC:  write R0 to input buffer of cell at address R1,
           or, if template, write R0 to input buffers of all creatures within
           PutLimit who have the complementary get template)
get    (read R0 from input port)

Calculation: 8

inc    (increment R0)
dec    (decrement R0)
add    (R2 = R1 + R0)
sub    (R2 = R1 - R0)
zero   (zero R0)
not0   (flip low order bit of R0)
shl    (shift left all bits of R0)
not    (flip all bits of R0)

Instruction Pointer Manipulation: 5

ifz  (if   R1 == 0 execute next instruction, otherwise, skip it)
ifZ  (if flag == 1 execute next instruction, otherwise, skip it)
jmp  (jump to template, or if no template jump to address in R0)
jmpb (jump back to template, or if no template jump back to address in R0)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr    (search outward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset +- R0)
adrb   (search backward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset - R0)
adrf   (search forward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset + R0)
mal    (allocate amount of space specified in R0, prefer address at R1,
           if R1 < 0 use better fit, place address of allocated block in R0)
divide (cell division, the IP is offset by R0 into the daughter cell, the
           values in the four CPU registers are transferred from mother to
           daughter, but not the stack.  If !R1, eject genome from soup)

Total: 32 instructions

The following array defines the associations of opcodes, mnemonics,
executable functions, and decode functions (in that order) used to
implement instruction set #1.  These associations are determined by
the opcode1.map file.

    {0, "nop0", nop, pnop, "", ""},            /* no decode args */
    {0, "nop1", nop, pnop, "", ""},            /* no decode args */
    {0, "A", regorder, dec1s, "a", ""},        /* "a" */
    {0, "B", regorder, dec1s, "b", ""},        /* "b" */
    {0, "C", regorder, dec1s, "c", ""},        /* "c" */
    {0, "D", regorder, dec1s, "d", ""},        /* "d" */
    {0, "movdd", movdd, dec1d1s, "ab", "H"},   /* "rr" */
    {0, "movdi", movdi, pmovdi, "ab", "H"},    /* "rr" */
    {0, "movid", movid, pmovid, "ba", "H"},    /* "rr" */
    {0, "movii", movii, pmovii, "ab", "H"},    /* "rr" */
    {0, "push", push, dec1s, "a", "H"},        /* "r" */
    {0, "pop", pop, dec1d, "a", "H"},          /* "r" */
    {0, "puticc", puticc, pputicc, "ab", "H"}, /* "rr" */
    {0, "get", get, dec1d1s, "a", "H"},        /* "r" */
    {0, "inc", math, dec1d1s, "aa", "H"},      /* "rr" */
    {0, "dec", math, dec1d1s, "aa", "H"},      /* "rr" */
    {0, "add", math, dec1d2s, "cba", "H"},     /* "rrr" */
    {0, "sub", math, dec1d2s, "cba", "H"},     /* "rrr" */
    {0, "zero", movdd, dec1d1s, "aa", "H"},    /* "rr" */
    {0, "shl", shl, dec1d, "a", "H"},          /* "r" */
    {0, "not0", not0, dec1d, "a", "H"},        /* "r" */
    {0, "not", not, dec1d, "a", "H"},          /* "r" */
    {0, "ifz", skip, dec2s, "ac", "H"},        /* "r#" */
    {0, "ifZ", skip, dec2s, "ab", ""},         /* "r#" */
    {0, "jmpo", adr, decjmp, "a", "H"},        /* "r" */
    {0, "jmpb", adr, decjmp, "a", "H"},        /* "r" */
    {0, "call", tcall, ptcall, "", ""},        /* no decode args */
    {0, "adro", adr, decadr, "abca", "H"},     /* "rr  " */
    {0, "adrb", adr, decadr, "abca", "H"},     /* "rr  " */
    {0, "adrf", adr, decadr, "abca", "H"},     /* "rr  " */
    {0, "mal", malchm, dec1d3s, "aa b", "H"},  /* "rr r" *
    {0, "divide", divide, dec2s, "ab", "H"},   /* "rr" */

******************************************************************************

Instset #2  Based on a design suggested and implemented by Tom Ray.  This
    includes certain features of the RPN Hewlett-Packard calculator.

    This instruction set operates on a CPU based on the following definitions:

#define STACK_SIZE 10
#define ALOC_REG 8
#define NUMREG 4         /* NUMREG = ALOC_REG */
#define GETBUFSIZ 3
#define PUTBUFSIZ 3

typedef struct {         /* structure for registers of virtual CPU */
    Reg re[ALOC_REG];    /* array of registers */
    Reg ip;              /* instruction pointer */
    Reg sp;              /* stack pointer */
    Reg st[STACK_SIZE];  /* stack */
    I8s fl;              /* flag */
    Reg gb[GETBUFSIZ+3]; /* input buffer */
    Reg pb[PUTBUFSIZ+3]; /* output buffer */
} Cpu;

The four registers in the re[] array are referred to as follows:
AX = re[0], BX = re[1], CX = re[2], DX = re[3].

No Operations: 2

nop0
nop1

Memory Movement: 11

rollu  (roll registers up:   AX = DX, BX = AX, CX = BX, DX = CX)
rolld  (roll registers down: AX = BX, BX = CX, CX = DX, DX = AX)
enter  (AX = AX, BX = AX, CX = BX, DX = CX, DX is lost)
exch   (AX = BX, BX = AX)
movdi  (move from BX to ram [AX])
movid  (move from ram [BX] to AX)
movii  (move from ram [BX] to ram [AX])
push   (push AX onto stack)
pop    (pop from stack into AX)
put    (write AX to output buffer, three modes:
```

```
          #ifndef ICC: write AX to own output buffer
          #ifdef ICC:  write AX to input buffer of cell at address BX,
            or, if template, write AX to input buffers of all creatures within
            PutLimit who have the complementary get template)
get    (read AX from input buffer)

Calculation: 9

inc    (increment AX)
dec    (decrement AX)
add    (AX = BX + AX, BX = CX, CX = DX))
sub    (AX = BX - AX, BX = CX, CX = DX))
zero   (zero AX)
not0   (flip low order bit of AX)
not    (flip all bits of AX)
shl    (shift left all bits of AX)
rand   (place random number in AX)

Instruction Pointer Manipulation: 5

ifz  (if   AX == 0 execute next instruction, otherwise, skip it)
ifZ (if flag == 1 execute next instruction, otherwise, skip it)
jmp  (jump to template, or if no template jump to address in AX)
jmpb (jump back to template, or if no template jump back to address in AX)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr    (search outward for template, put address in AX, template size in BX,
           and offset in CX, start search at offset +- BX)
adrb   (search backward for template, put address in AX, template size in BX,
           and offset in CX, start search at offset - BX)
adrf   (search forward for template, put address in AX, template size in BX,
           and offset in CX, start search at offset + BX)
mal    (allocate amount of space specified in BX, prefer address at AX,
           if AX < 0 use better fit, place address of allocated block in AX)
divide (cell division, the IP is offset by AX into the daughter cell, the
           values in the four CPU registers are transferred from mother to
           daughter, but not the stack. If !CX genome will be ejected from
           the simulator)

Total: 32 instructions

The following array defines the associations of opcodes, mnemonics,
executable functions, and decode functions (in that order) used to
implement instruction set #2.  These associations are determined in
the opcode2.map file.

    {0, "nop0", nop, pnop, "", ""},           /* no decode args */
    {0, "nop1", nop, pnop, "", ""},           /* no decode args */
    {0, "rollu", rollu, pnop, "", ""},        /* no decode args */
    {0, "rolld", rolld, pnop, "", ""},        /* no decode args */
    {0, "enter", enter, pnop, "", ""},        /* no decode args */
    {0, "exch", exch, dec2s, "ab", ""},       /* "rr" */
    {0, "movdi", movdi, pmovdi, "ab", ""},    /* "rr" */
    {0, "movid", movid, pmovid, "ba", ""},    /* "rr" */
    {0, "movii", movii, pmovii, "ab", ""},    /* "rr" */
    {0, "push", push, dec1s, "a", ""},        /* "r" */
    {0, "pop", pop, dec1d, "a", "P"},         /* "r" */
    {0, "puticc", puticc, pputicc, "ab", ""}, /* "rr" */
    {0, "get", get, dec1d1s, "a", ""},        /* "r" */
    {0, "inc", math, dec1d1s, "aa", ""},      /* "rr" */
    {0, "dec", math, dec1d1s, "aa", ""},      /* "rr" */
    {0, "add", math, dec1d2s, "aba", "P"},    /* "rrr" */
    {0, "sub", math, dec1d2s, "aba", "P"},    /* "rrr" */
    {0, "zero", movdd, dec1d1s, "aa", "P"},   /* "rr" */
    {0, "shl", shl, dec1d, "a", ""},          /* "r" */
    {0, "not0", not0, dec1d, "a", ""},        /* "r" */
    {0, "not", not, dec1d, "a", ""},          /* "r" */
    {0, "rand", movdd, dec1d1s, "aa", "P"},   /* "rr" */
    {0, "ifz", skip, dec2s, "ac", ""},        /* "r#" */
    {0, "ifZ", skip, dec2s, "ab", ""},        /* "r#" */
    {0, "jmpo", adr, decjmp, "a", ""},        /* "r" */
    {0, "jmpb", adr, decjmp, "a", ""},        /* "r" */
    {0, "call", tcall, ptcall, "", ""},       /* no parse args */
    {0, "adro", adr, decadr, "abcb", "P"},    /* "rr  " */
    {0, "adrb", adr, decadr, "abcb", "P"},    /* "rr  " */
    {0, "adrf", adr, decadr, "abcb", "P"},    /* "rr  " */
    {0, "mal", malchm, dec1d3s, "ab a", "P"}, /* "rr r" */
    {0, "divide", divide, dec2s, "ab", ""},   /* "rr" */

******************************************************************************

Instset #3 Based on a design suggested by Walter Tackett of Hughes Aircraft,
    and implemented by Tom Ray.  The special features of this instruction
    set are that all movement between registers of the CPU takes place via
    push and pop through the stack.  Also, all indirect addressing involves
    an offset from the address in the CX register.  Also, the CX register
    is where most calculations take place.

    This instruction set operates on a CPU based on the following definitions:

#define STACK_SIZE 10
#define ALOC_REG 8
#define NUMREG 4         /* NUMREG = ALOC_REG */
#define GETBUFSIZ 3
#define PUTBUFSIZ 3

typedef struct {         /* structure for registers of virtual CPU */
    Reg re[ALOC_REG];    /* array of registers */
    Reg ip;              /* instruction pointer */
    Reg sp;              /* stack pointer */
    Reg st[STACK_SIZE];  /* stack */
    I8s fl;              /* flag */
    Reg gb[GETBUFSIZ+3]; /* input buffer */
    Reg pb[PUTBUFSIZ+3]; /* output buffer */
} Cpu;

The four registers in the re[] array are referred to as follows:
AX = re[0], BX = re[1], CX = re[2], DX = re[3].

No Operations: 2

nop0
nop1

Memory Movement: 13

movdi  (move from BX to ram [AX + CX])
movid  (move from ram [BX + CX] to AX)
movii  (move from ram [BX + CX] to ram [AX + CX])
pushax (push AX onto stack)
pushbx (push BX onto stack)
pushcx (push CX onto stack)
pushdx (push DX onto stack)
popax  (pop from stack into AX)
popbx  (pop from stack into BX)
popcx  (pop from stack into CX)
popdx  (pop from stack into DX)
put    (write DX to output buffer, three modes:
```

```
          #ifndef ICC: write DX to own output buffer
          #ifdef ICC:  write DX to input buffer of cell at address CX,
            or, if template, write DX to input buffers of all creatures within
            PutLimit who have the complementary get template)
get    (read DX from input port)

Calculation: 7

inc    (increment CX)
dec    (decrement CX)
add    (CX = CX + DX)
sub    (CX = CX - DX)
zero   (zero CX)
not0   (flip low order bit of CX)
shl    (shift left all bits of CX)

Instruction Pointer Manipulation: 5

ifz    (if   CX == 0 execute next instruction, otherwise, skip it)
ifZ    (if flag == 1 execute next instruction, otherwise, skip it)
jmp    (jump to template, or if no template jump to address in BX)
jmpb   (jump back to template, or if no template jump back to address in BX)
call   (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr    (search outward for template, put address in AX, template size in DX,
            and offset in CX, start search at offset +- CX)
adrb   (search backward for template, put address in AX, template size in DX,
            and offset in CX, start search at offset - CX)
adrf   (search forward for template, put address in AX, template size in DX,
            and offset in CX, start search at offset + CX)
mal    (allocate amount of space specified in CX, prefer address at AX,
            if AX < 0 use better fit, place address of allocated block in AX)
divide (cell division, the IP is offset by CX into the daughter cell, the
            values in the four CPU registers are transferred from mother to
            daughter, but not the stack.  If !DX genome will be ejected from
            the simulator)

Total: 32 instructions

The following array defines the associations of opcodes, mnemonics,
executable functions, and parsing functions (in that order) used to
implement instruction set #3.  These associations are determined by
the opcode3.map file.

Segment registers: c
Destination registers: a
Source registers: b
    {0, "nop0", nop, pnop, "", ""},           /* no decode args */
    {0, "nop1", nop, pnop, "", ""},           /* no decode args */
    {0, "movdi", movdi, pmovdi, "abc", "FTG"},/* "rr" */
    {0, "movid", movid, pmovid, "bac", "F"},  /* "rr" */
    {0, "movii", movii, pmovii, "abc", "FTG"},/* "rr" */
    {0, "pushA", push, dec1s, "a", ""},       /* "a" */
    {0, "pushB", push, dec1s, "b", ""},       /* "b" */
    {0, "pushC", push, dec1s, "c", ""},       /* "c" */
    {0, "pushD", push, dec1s, "d", ""},       /* "d" */
    {0, "popA", pop, dec1d, "a", ""},         /* "a" */
    {0, "popB", pop, dec1d, "b", ""},         /* "b" */
    {0, "popC", pop, dec1d, "c", ""},         /* "c" */
    {0, "popD", pop, dec1d, "d", ""},         /* "d" */
    {0, "puticc", puticc, pputicc, "dc", ""}, /* "rr" */
    {0, "get", get, dec1d1s, "d", ""},        /* "r" */
    {0, "inc", math, dec1d1s, "cc", ""},      /* "rr" */
    {0, "dec", math, dec1d1s, "cc", ""},      /* "rr" */
    {0, "add", math, dec1d2s, "ccd", ""},     /* "rrr" */
    {0, "sub", math, dec1d2s, "ccd", ""},     /* "rrr" */
    {0, "zero", movdd, dec1d1s, "cc", ""},    /* "rr" */
    {0, "shl", shl, dec1d, "c", ""},          /* "r" */
    {0, "not0", not0, dec1d, "c", ""},        /* "r" */
    {0, "ifz", skip, dec2s, "cc", ""},        /* "r#" */
    {0, "ifZ", skip, dec2s, "ab", ""},        /* "r#" */
    {0, "jmpo", adr, decjmp, "b", ""},        /* "r" */
    {0, "jmpb", adr, decjmp, "b", ""},        /* "r" */
    {0, "call", tcall, ptcall, "", ""},       /* no decode args */
    {0, "adro", adr, decadr, "adcc", ""},     /* "rr  " */
    {0, "adrb", adr, decadr, "adcc", ""},     /* "rr  " */
    {0, "adrf", adr, decadr, "adcc", ""},     /* "rr  " */
    {0, "mal", malchm, dec1d3s, "ac a", ""},  /* "rr r" */
    {0, "divide", divide, dec2s, "cd", ""},   /* "rr" */

*****************************************************************************
```

8.4) Details of Set 0

What follows is a more detailed documentation of instruction set 0:

In general, the ax and bx registers are used to hold addresses, which refer to locations in the soup. Values contained in the ax and bx registers are maintained as positive integers, modulus SoupSize. The cx and dx registers are generally used to hold numbers, which may be either positive or negative. Values contained in the cx and dx registers are forced to remain in the range -SoupSize to SoupSize. Any operation which causes the values in the cx or dx registers to stray out of this range will cause the value in the register to be reset to zero.

Many of the instructions will fail under certain conditions, which are specified for each instruction in the following discussion. Any time an instruction fails, it does nothing, other than to increment the instruction pointer, decrement the instruction bank, and set the flag register (to the value 1). Any time and instruction does not fail, it clears the flag register (to the value 0). All instructions clear the flag on success, and set the flag on failure. Every instruction decrements the instruction bank by one, regardless of success or failure. Unless otherwise indicated, all instructions increment the instruction pointer by one (modulus SoupSize).

What follows is a list of the thirty-two instructions of set 0, along with details of their behavior:

```
{0, "nop0", nop, pnop, "", ""}, /* do nothing */
{0, "nop1", nop, pnop, "", ""},
```

These two instructions are no-ops, they do nothing, other than to increment the instruction pointer one place. There are no conditions under which these instructions fail.

```
{0, "not0", not0, dec1d, "c", ""}, /* flip low order bit of cx */
```

This instruction flips the low order bit of the cx register. There are no conditions under which this instruction fails.

```
{0, "shl", shl, dec1d, "c", ""}, /* shift left all bits of cx */
```

This instruction shifts all bits of the cx register one position to the left, replacing the rightmost (low order) bit with a zero (this is a binary multiply by two). There are no conditions under which this instruction fails.

```
{0, "zero", movdd, dec1d1s, "cc", ""}, /* cx = 0 */
```

This instruction sets the cx register to zero. It also increments the instruction pointer one place. There are no conditions under which this instruction fails.

```
{0, "ifz", skip, dec2s, "cc", ""},
       /* execute next instruction only if cx == 0 */
```

This instruction will increment the instruction pointer by one if the value in the cx register is zero, or by two if the cx register contains any other value (this means that the instruction following this one will be executed only if the value in the cx register is zero). There are no conditions under which this instruction fails.

```
{0, "subCAB", math, dec1d2s, "cab", ""}, /* cx = ax - bx */
```

This instruction subtracts the value in the bx register from the value in the ax register, placing the result in the cx register. There are no conditions under which this instruction fails.

```
{0, "subAAC", math, dec1d2s, "aac", ""}, /* ax = ax - cx */
```

This instruction subtracts the value in the cx register from the value in the ax register, placing the result in the ax register. There are no conditions under which this instruction fails.

```
{0, "incA", math, dec1d1s, "aa", ""}, /* ax++ */
```

This instruction increments (adds one to) the value in the ax register, placing the result in the ax register. There are no conditions under which this instruction fails.

```
{0, "incB", math, dec1d1s, "bb", ""}, /* bx++ */
```

This instruction increments (adds one to) the value in the bx register, placing the result in the bx register. There are no conditions under which this instruction fails.

```
{0, "decC", math, dec1d1s, "cc", ""}, /* cx-- */
```

This instruction decrements (subtracts one from) the value in the cx register, placing the result in the cx register. There are no conditions under which this instruction fails.

```
{0, "incC", math, dec1d1s, "cc", ""}, /* cx++ */
```

This instruction increments (adds one to) the value in the cx register, placing the result in the cx register. There are no conditions under which this instruction fails.

```
{0, "pushA", push, dec1s, "a", ""}, /* push ax onto stack */
```

This instruction causes the value in the ax register to be pushed onto the stack, and the stack pointer to be incremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "pushB", push, dec1s, "b", ""}, /* push bx onto stack */
```

This instruction causes the value in the bx register to be pushed onto the stack, and the stack pointer to be incremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "pushC", push, dec1s, "c", ""}, /* push cx onto stack */
```

This instruction causes the value in the cx register to be pushed onto the stack, and the stack pointer to be incremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "pushD", push, dec1s, "d", ""}, /* push dx onto stack */
```

This instruction causes the value in the dx register to be pushed onto the stack, and the stack pointer to be incremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "popA", pop, dec1d, "a", ""}, /* pop ax off of stack */
```

This instruction causes the value at the top of the stack to be popped into the ax register and the stack pointer to be decremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "popB", pop, dec1d, "b", ""}, /* pop bx off of stack */
```

This instruction causes the value at the top of the stack to be popped into the bx register and the stack pointer to be decremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "popC", pop, dec1d, "c", ""}, /* pop cx off of stack */
```

This instruction causes the value at the top of the stack to be popped into the cx register and the stack pointer to be decremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "popD", pop, dec1d, "d", ""}, /* pop dx off of stack */
```

This instruction causes the value at the top of the stack to be popped into the dx register and the stack pointer to be decremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "jmpo", adr, decjmp, "b", ""}, /* outward template jump */
```

This instruction causes the instruction pointer to be redirected to the instruction following the nearest occurrence of the template complementary to the template that follows the jmp instruction. The template is defined as the group of consecutive nops (nop0 or nop1) that immediately follow the jmp instruction. The size of the template is just the number of consecutive nops that follow the jmp instruction. The target template can be larger than the source template (the entire source template must be matched, but it may match to a sub-template in the target template).

This instruction is a bi-directional jump, which means that it will search both forward and backward in memory for the complementary template. Let a equal the address of the first instruction of the template, and let s equal the size of the template. The forward search starts at the address a + s + 1, and the backward search starts at the address a - s - 1. The search looks in the forward direction first, then alternately looks ahead or back one additional step until the complementary template is found, or the search limit (Search_limit) is reached. The template searches wrap around the ends of the soup.

The instruction will fail if the size of the template following the jmp instruction is less than MinTemplSize, or greater than SoupSize, or if a complementary template is not found within the search radius, Search_limit. On failure, the instruction pointer is moved past the template used by the jmp instruction: ce->c.ip = ce->c.ip + s + 1.

```
{0, "jmpb", adr, decjmp, "b", ""}, /* backward template jump */
```

This instruction causes the instruction pointer to be redirected to the instruction following the nearest occurrence (backwards in the soup) of the template complementary to the template that follows the jmp instruction. The template is defined as the group of consecutive nops (nop0 or nop1) that immediately follow the jmp instruction. The size of the template is just the number of consecutive nops that follow the jmp instruction. The target template can be larger than the source template (the entire source template must be matched, but it may match to a sub-template in the target template).

This instruction is a uni-directional jump, which means that it will search in only one direction (in this case, only backward) in memory for the complementary template. Let a equal the address of the first instruction of the template, and let s equal the size of the template. The backward search starts at the address a - s - 1. The search looks in successive steps backwards until the complementary template is found, or the search limit (Search_limit) is reached. The template searches wrap around the ends of the soup.

The instruction will fail if the size of the template following the jmp instruction is less than MinTemplSize, or greater than SoupSize, or if a complementary template is not found within the search radius, Search_limit. On failure, the instruction pointer is moved past the template used by the jmp instruction: ce->c.ip = ce->c.ip + s + 1.

```
{0, "call", tcall, ptcall, "", ""},
          /* push ip to stack, outward template jump */
```

This instruction behaves identically to the jmp instruction, with the one difference that the address following the tempate following the call instruction is also pushed onto the stack (this is essentially the address of the instruction pointer plus one meaningful instruction). On failure, the instruction pointer is moved past the template used by the jmp instruction: ce->c.ip = ce->c.ip + s + 1, and the ip will not be pushed onto the stack. If the call instruction is not followed by a template, it will push the address of the instruction pointer + 1 onto the stack.

```
{0, "ret", pop, dec1d, "", ""}, /* pop ip from stack */
```

This instruction causes the value at the top of the stack to be popped into the instruction pointer and the stack pointer to be decremented (modulus STACK_SIZE). There are no conditions under which this instruction fails.

```
{0, "movDC", movdd, dec1d1s, "dc", ""}, /* dx = cx */
```

This instruction copies the contents of the cx register into the dx register, leaving the value in cx intact. The flaw may cause the source register to actually be bx or dx and the destination register to actually be cx or ax. There are no conditions that can cause this instruction to fail.

```
{0, "movBA", movdd, dec1d1s, "ba", ""}, /* bx = ax */
```

This instruction copies the contents of the ax register into the bx register, leaving the value in ax intact. The flaw may cause the source register to actually be bx or dx and the destination register to actually be cx or ax. There are no conditions that can cause this instruction to fail.

```
{0, "movii", movii, pmovii, "ab", ""}, /* [bx] = [ax] */
```

This instruction copies one instruction in the soup to another location in the soup. The source instruction is at the address contained in the bx register, and the destination is the address contained in the ax register.

This instruction could fail under the following circumstances: a) if the source and destination addresses are the same, b) if the destination address is not owned by this creature and is write protected, c) if the source address is not owned by this creature and is read protected, d) if either the source or destination addresses are outside the soup.

Also, if the destination address is in the daughter, it will perform the following operation: ce->d.mov_daught++. If the destination address is not in the daughter cell, it will make a call to MutBookeep(is.dval), since this could concievably make a genetic change in another creature (if the other creature were not write protected, or if the ``other'' creature were actually the one doing the writing).

```
{0, "adro", adr, decadr, "ac  ", ""},
          /* search outward for template, return adr in ax */
```

This instruction causes the ax register to contain the address of the instruction following the nearest occurrence of the template complementary to the template that follows the adr instruction. The template is defined as the group of consecutive nops (nop0 or nop1) that immediately follow the adr instruction. The size of the template is just the number of consecutive nops that follow the adr instruction. The size of the template is placed in the cx register. The target template can be larger than the source template (the entire source template must be matched, but it may match to a sub-template in the target template).

This instruction is a bi-directional search, which means that it will search both forward and backward in memory for the complementary template. Let a equal the address of the first instruction of the template, and let s equal the size of the template. The forward search starts at the address a + s + 1, and the backward search starts at the address a - s - 1. The search looks in the forward direction first, then alternately looks ahead or back one additional step until the complementary template is found, or the search limit (Search_limit) is reached. The template searches wrap around the ends of the soup. The instruction pointer is moved past the template used by the adr instruction: ce->c.ip = ce->c.ip + s + 1.

The instruction will fail if the size of the template following the adr instruction is less than MinTemplSize, or greater than SoupSize, or if a complementary template is not found within the search radius, Search_limit. On failure and the cx register is not altered.

```
{0, "adrb", adr, decadr, "ac  ", ""},
          /* search backward for template, rtrn adr in ax */
```

This instruction causes the ax register to contain the address of the instruction following the nearest occurrence (backwards in the soup) of the template complementary to the template that follows the adrb instruction. The template is defined as the group of consecutive nops (nop0 or nop1) that immediately follow the adrb instruction. The size of the template is just the number of consecutive nops that follow the adrb instruction. The size of the template is placed in the cx register. The target template can be larger than the source template (the entire source template must be matched, but it may match to a sub-template in the target template).

This instruction is a uni-directional search, which means that it will search in only one direction (in this case, only backward) in memory for the complementary template. Let a equal the address of the first instruction of the template, and let s equal the size of the template. The backward search starts at the address a - s - 1. The search looks in successive steps backwards until the complementary template is found, or the search limit (Search_limit) is reached. The template searches wrap around the ends of the soup. The instruction pointer is moved past the template used by the adrb instruction: ce->c.ip = ce->c.ip + s + 1.

The instruction will fail if the size of the template following the adrb instruction is less than MinTemplSize, or greater than SoupSize, or if a complementary template is not found within the search radius, Search_limit. On failure the cx register is not altered.

```
{0, "adrf", adr, decadr, "ac  ", ""},
          /* search forward for template, rtrn adr in ax */
```

This instruction causes the ax register to contain the address of the instruction following the nearest occurrence (forwards in the soup) of the template complementary to the template that follows the adrf instruction. The template is defined as the group of consecutive nops (nop0 or nop1) that immediately follow the adrf instruction. The size of the template is just the number of consecutive nops that follow the adrf instruction. The size of the template is placed in the cx register. The target template can be larger than the source template (the entire source template must be matched, but it may match to a sub-template in the target template).

This instruction is a uni-directional search, which means that it will search in only one direction (in this case, only forward) in memory for the complementary template. Let a equal the address of the first instruction of the template, and let s equal the size of the template. The forward search starts at the address a + s + 1. The search looks in successive steps forwards until the complementary template is found, or the search limit (Search_limit) is reached. The template searches wrap around the ends of the soup. The instruction pointer is moved past the template used by the adrf instruction: ce->c.ip = ce->c.ip + s + 1.

The instruction will fail if the size of the template following the adrf instruction is less than MinTemplSize, or greater than SoupSize, or if a complementary template is not found within the search radius, Search_limit. On failure the cx register is not altered.

```
{0, "mal", malchm, dec1d3s, "ac a", ""},
          /* allocate & chmod space for a new cell */
```

This instruction requests memory space in the soup from the operating system. The amount of space requested is specified by the value in the cx register. The address of the allocated block is returned in the ax register. The memory allocated in this way is considered to belong to the creature whose CPU executed this instruction. The memory protections specified by the MemModeMine and MemModeProt variables apply. This instruction will increment the instruction pointer by one. If this instruction is executed successfully, this creature will move down the reaper queue one position.

The operating system searches for a block of free memory of the size requested, and will invoke the reaper if such a block is not available. The block of memory may be located on a first fit basis by beginning the search at the beginning of memory, or using a better fit method, or the block may be located at a preferred location to within a tolerance (see the MalMode, MalReapTol, and MalTol variables described in the section on SOUP_IN PARAMETERS). Note that the execution of this instruction usually causes the reaper to kill in order to free space to meet the request. However, the reaper will never kill the creature making the request. The allocated block may not wrap around the end of the soup.

This instruction can fail under the following circumstances: a) if the amount of memory requested is less than or equal to zero (note that a request of 1 may be converted into a request of 0 by a flaw), b) if the creature already owns a second block of memory whose size is exactly the amount of memory requested (in other words, if the creature makes two successive calls to mal(), requesting the same amount of memory, without an intervening call to divide()), c) if the amount of memory requested is more than MaxMalMult times the size of the creature making the request (the default value of MaxMalMult is 3).

```
{0, "divide", divide, dec2s, "aa", "C"},
          /* give life to new cell, put in queues */
```

This instruction causes the mother cell to loose her write privelages on the space of the daughter cell, and causes the daughter cell to be entered into both the reaper and slicer queues. The daughter enters the bottom of the reaper queue, and enters the slicer queue just behind the mother (so the daughter will be the last cell to be reaped and to get another slice). This instruction will increment the instruction pointer by one. If this instruction is executed successfully, this creature will move down the reaper queue one position.

This instruction can fail under the following circumstances: a) if the size of the daughter cell is less than soup_in variable MinCellSize b) if the number of instructions written into the space of the daughter cell by the mother cell (ce->d.mov_daught) is less than the soup_in variable MovPropThrDiv times the size of the daughter cell, c) if the soup_in variable DivSameSize is non-zero, and the daughter cell is not the same size as the mother cell, d) if the soup_in variable DivSameGen is non-zero and the daughter cell is not the same size, or does not have exactly the same sequence as the mother cell.

8.5) New Features in Sets 1 through 3

Instruction sets one through three are essentially identical except for the methods used to move data between the registers of the CPU. Set one allows the order of the registers to be freely rearranged, and provides an instruction, movdd() to move data directly between two registers. Set two uses the reverse Polish notation method, in which the registers can be rolled up or down, and the values in the lower two registers can be exchanged. Set three requires all inter-register moves to pass through the stack.

Each of the new sets includes a movdi() instruction to move data from a register into the soup, and a movid instruction to move data from the soup into a register.

Each new set provides I/O functions in the form of put() and get() instructions which write to or read from buffers. The get() instruction reads a value from the input buffer and moves it into a CPU register. The input buffer maintains a pointer to the next input value to be read from the buffer, a pointer to the next input value to be written to the buffer, and a count of unread input values:

```
 ce->c.gb[GETBUFSIZ]     == pointer to next input value to be read
 ce->c.gb[GETBUFSIZ + 1] == pointer to next input value to be written
 ce->c.gb[GETBUFSIZ + 2] == number of unread input values
```

The output buffer maintains a pointer to the next output value to be written to the buffer, a pointer to the next output value to be read from the buffer, and a count of unread output values:

```
 ce->c.pb[PUTBUFSIZ]     == pointer to next output value to be written
 ce->c.pb[PUTBUFSIZ + 1] == pointer to next output value to be read
 ce->c.pb[PUTBUFSIZ + 2] == number of unread output values
```

The put() instruction operates in two modes, which are implemented by two versions: put() and puticc(). Puticc() writes to the input buffer(s) of other cell(s). In this case, if the puticc() instruction is followed by a template, puticc() will write to the input buffer(s) of any cells within the put radius, PutLimit, which have a get instruction followed by a complementary template (this is an analog to hormonal communication). If puticc() is not followed by a template, then it writes to the input buffer of the cell located at the address specified in the relevant register of the cell executing the puticc().

The plain version, put() writes to its own output buffer. This mode is provided so that the user can communicate with the cells. This could be used for example to get them to do useful work by giving them data by writing to their input buffer, letting them get the data and compute on it, then letting them write their results back to you in the output buffer (the creatures can also use this facility to pray). In order to assist with communication between the user and the creatures, three additional functions are provided in the instruct.c module:

```
I8s ReadFPut(ce, value)   /* for god to read the data in the output buffer */
void Write2Get(ce, value) /* place value in input buffer of cell ce */
void Broad2Get(value)     /* broadcast value to input buffer of all cells */
```

Each new instruction set includes the basic calculations: inc(), dec(), add() and sub(). The zero() instruction sets a register to zero. The not0() instruction flips the low order bit of a register, and the not() instruction flips all bits of a register. The shl() instruction shifts all bits of a register one bit to the left, placing a zero in the low order bit (multiply by two). Set two also provides rand() which places a random number in the AX register.

In addition to ifz() which tests if a register is zero, the new sets provide ifZ() which tests it a flag is set. This latter can be used to test in an instruction has failed.

In sets one through three, the jmpo() and jmpb() instructions will jump to a complementary template, if the jump instruction is followed by a template, but if there is no template, they will jump to the address specified in a register. Similarly, the call instruction will push the instruction pointer + 1 onto the stack, and if a template is present, will jump to the nearest complementary template.

In sets one through three, the address instructions adro(), adrb() and adrf() have two new features: 1) they return the offset distance from the source template to where they found the target template. 2) they start the search at an offset from the source template, specified in a register. This provides the facility to repeatedly call the address instructions in order to find target templates beyond the first matching one.

In sets one through three, the mal() instruction has the option of specifying the preferred address in a register, or of using a better fit mode.

In sets one through three, the divide instruction causes the contents of the mother's CPU registers to be transferred to the daughter's CPU registers. Also, the instruction pointer of the daughter will start with an offset into the daughter cell that is specified in a register of the mother's CPU. This allows the mother to force the daughter to express a different sub-set of the genome from that which was expressed by the mother (the mother can force the daughter cell to differentiate). If a certain register of the mother is not zero, the genome of the daughter cell will be ejected from the soup. This makes it possible for the daughter cell to emmigrate to another soup, if there is one. When ejected from the soup, the space occupied by the daughter cell is freed, and the code is set to zero in that space.

8.6) Set 8

Set 8 is an experimental six bit haploid instruction set defined by the opcode8.map file (see below). This instruction set was developed by Kurt Thearling in order to experiment with the evolution of mulit-cellular digital organisms.

```
    {0, "nop0", nop, pnop, "", ""},              /* no decode args */
    {0, "nop1", nop, pnop, "", ""},              /* no decode args */
    {0, "not0", not0, dec1d, "c", ""},           /* "r" */
    {0, "shl", shl, dec1d, "c", ""},             /* "r" */
    {0, "zero", movdd, dec1d1s, "cc", ""},       /* "rr" */
    {0, "ifz", skip, dec2s, "cc", ""},           /* "r#" */
    {0, "subCAB", math, dec1d2s, "cab", ""},     /* "rrr" */
    {0, "subAAC", math, dec1d2s, "aac", ""},     /* "rrr" */
    {0, "incA", math, dec1d1s, "aa", ""},        /* "aa" */
    {0, "incB", math, dec1d1s, "bb", ""},        /* "bb" */
    {0, "decC", math, dec1d1s, "cc", ""},        /* "cc" */
    {0, "incC", math, dec1d1s, "cc", ""},        /* "cc" */
    {0, "pushA", push, dec1s, "a", ""},          /* "a" */
    {0, "pushB", push, dec1s, "b", ""},          /* "b" */
    {0, "pushC", push, dec1s, "c", ""},          /* "c" */
    {0, "pushD", push, dec1s, "d", ""},          /* "d" */
    {0, "popA", pop, dec1d, "a", ""},            /* "a" */
    {0, "popB", pop, dec1d, "b", ""},            /* "b" */
    {0, "popC", pop, dec1d, "c", ""},            /* "c" */
    {0, "popD", pop, dec1d, "d", ""},            /* "d" */
    {0, "jmpo", adr, decjmp, "b", ""},           /* "r" */
    {0, "jmpb", adr, decjmp, "b", ""},           /* "r" */
    {0, "call", tcall, ptcall, "", ""},          /* no decode args */
    {0, "ret", pop, dec1d, "", ""},              /* no decode args */
    {0, "movDC", movdd, dec1d1s, "dc", ""},      /* "dc" */
    {0, "movBA", movdd, dec1d1s, "ba", ""},      /* "ba" */
    {0, "movii", movii, pmovii, "ab", ""},       /* "rr" */
    {0, "adro", adr, decadr, "ac  ", ""},        /* "rr  " */
    {0, "adrb", adr, decadr, "ac  ", ""},        /* "rr  " */
    {0, "adrf", adr, decadr, "ac  ", ""},        /* "rr  " */
    {0, "mal", malchm, dec1d3s, "ac a", ""},     /* "rr r" */
    {0, "divide", divide, dec2s, "aa", "C"},     /* "rr" */
    {0, "split", split, dec1d1s, "dd", ""},      /* "rr" */
    {0, "join", join, pnop, "", ""},             /* no decode args */
    {0, "not0", not0, dec1d, "c", ""},           /* "r" */
    {0, "shr", shr, dec1d, "c", ""},             /* "r" */
    {0, "zeroD", movdd, dec1d1s, "dd", ""},      /* "dd" */
    {0, "ifz", skip, dec2s, "cc", ""},           /* "r#" */
    {0, "offAACD", offset, dec1d3s, "aacd", ""}, /* "rrrr" */
    {0, "offBBCD", offset, dec1d3s, "bbcd", ""}, /* "rrrr" */
    {0, "incA", math, dec1d1s, "aa", ""},        /* "aa" */
    {0, "incB", math, dec1d1s, "bb", ""},        /* "bb" */
    {0, "decC", math, dec1d1s, "cc", ""},        /* "cc" */
    {0, "incC", math, dec1d1s, "cc", ""},        /* "cc" */
    {0, "pushA", push, dec1s, "a", ""},          /* "a" */
    {0, "pushB", push, dec1s, "b", ""},          /* "b" */
    {0, "pushC", push, dec1s, "c", ""},          /* "c" */
    {0, "pushD", push, dec1s, "d", ""},          /* "d" */
    {0, "popA", pop, dec1d, "a", ""},            /* "a" */
    {0, "popB", pop, dec1d, "b", ""},            /* "b" */
    {0, "popC", pop, dec1d, "c", ""},            /* "c" */
    {0, "popD", pop, dec1d, "d", ""},            /* "d" */
    {0, "jmpo", adr, decjmp, "b", ""},           /* "r" */
    {0, "jmpb", adr, decjmp, "b", ""},           /* "r" */
    {0, "call", tcall, ptcall, "", ""},          /* no decode args */
    {0, "ret", pop, dec1d, "", ""},              /* no decode args */
    {0, "movDC", movdd, dec1d1s, "dc", ""},      /* "dc" */
    {0, "movBA", movdd, dec1d1s, "ba", ""},      /* "ba" */
    {0, "movii", movii, pmovii, "ab", ""},       /* "rr" */
    {0, "adro", adr, decadr, "ac  ", ""},        /* "rr  " */
    {0, "adrb", adr, decadr, "ac  ", ""},        /* "rr  " */
    {0, "adrf", adr, decadr, "ac  ", ""},        /* "rr  " */
    {0, "mal", malchm, dec1d3s, "ac a", ""},     /* "rr r" */
    {0, "divide", divide, dec2s, "aa", "C"},     /* "rr" */
```

In this instruction set there are six fully new instructions:

```
    {0, "split", split, dec1d1s, "dd", ""},      /* "rr" */
    {0, "join", join, pnop, "", ""},             /* no decode args */
    {0, "shr", shr, dec1d, "c", ""},             /* "r" */
    {0, "zeroD", movdd, dec1d1s, "dd", ""},      /* "dd" */
    {0, "offAACD", offset, dec1d3s, "aacd", ""}, /* "rrrr" */
    {0, "offBBCD", offset, dec1d3s, "bbcd", ""}, /* "rrrr" */
```

Split and join were created in order to allow a creature to change the number of processors that it has executing. Shr effects a divide by two operation. ZeroD zeros out the DX register. OffAACD and offBBCD are supposed to be used to calculate offsets from a base address.

In order to use this instruction set, Tierra must be compiled with PLOIDY defined as 1 (see the configur.h file). This set can be used with the si8 soup_in file, and the genomes contained in the gb8/ directory.

---

9) The Ancestor & Writing a Creature

9.1) The Ancestor

The ASCII assembler code file with comments, for the ancestor, is listed below. Below the listing I have some explanatory material.

```
**** begin genome file (note blank line at head of file)

format: 3  bits: 2156009669  EXsh    TCsh    TPs     MFs     MTd     MBh
genotype: 0080aaa  parent genotype: 0666god
1st_daughter:  flags: 0  inst: 827  mov_daught: 80          breed_true: 1
2nd_daughter:  flags: 0  inst: 809  mov_daught: 80          breed_true: 1
Origin: InstExe: 0,0  clock: 0  Thu Jan 01 -5:00:00 1970
MaxPropPop: 0.8306  MaxPropInst: 0.4239 mpp_time: 0,0
ploidy: 1  track: 0
comments: the ancestor, written by a human, mother of all other creatures.

track 0:

nop1    ; 110 01   0 beginning marker
nop1    ; 110 01   1 beginning marker
nop1    ; 110 01   2 beginning marker
nop1    ; 110 01   3 beginning marker
zero    ; 110 04   4 put zero in cx
not0    ; 110 02   5 put 1 in first bit of cx
shl     ; 110 03   6 shift left cx (cx = 2)
shl     ; 110 03   7 shift left cx (cx = 4)
movDC   ; 110 18   8 move cx to dx (dx = 4)
adrb    ; 110 1c   9 get (backward) address of beginning marker -> ax
nop0    ; 100 00  10 complement to beginning marker
nop0    ; 100 00  11 complement to beginning marker
nop0    ; 100 00  12 complement to beginning marker
nop0    ; 100 00  13 complement to beginning marker
subAAC  ; 110 07  14 subtract cx from ax, result in ax
```

```
movBA   ; 110 19  15 move ax to bx, bx now contains start address of mother
adrf    ; 110 1d  16 get (forward) address of end marker -> ax
nop0    ; 100 00  17 complement to end marker
nop0    ; 100 00  18 complement to end marker
nop0    ; 100 00  19 complement to end marker
nop1    ; 100 01  20 complement to end marker
incA    ; 110 08  21 increment ax, to include dummy instruction at end
subCAB  ; 110 06  22 subtract bx from ax to get size, result in cx
nop1    ; 110 01  23 reproduction loop marker
nop1    ; 110 01  24 reproduction loop marker
nop0    ; 110 00  25 reproduction loop marker
nop1    ; 110 01  26 reproduction loop marker
mal     ; 110 1e  27 allocate space (cx) for daughter, address to ax
call    ; 110 16  28 call template below (copy procedure)
nop0    ; 100 00  29 copy procedure complement
nop0    ; 100 00  30 copy procedure complement
nop1    ; 100 01  31 copy procedure complement
nop1    ; 100 01  32 copy procedure complement
divide  ; 110 1f  33 create independent daughter cell
jmpo    ; 110 14  34 jump to template below (reproduction loop)
nop0    ; 100 00  35 reproduction loop complement
nop0    ; 100 00  36 reproduction loop complement
nop1    ; 100 01  37 reproduction loop complement
nop0    ; 100 00  38 reproduction loop complement
ifz     ; 000 05  39 dummy instruction to separate templates
nop1    ; 110 01  40 copy procedure template
nop1    ; 110 01  41 copy procedure template
nop0    ; 110 00  42 copy procedure template
nop0    ; 110 00  43 copy procedure template
pushA   ; 110 0c  44 push ax onto stack
pushB   ; 110 0d  45 push bx onto stack
pushC   ; 110 0e  46 push cx onto stack
nop1    ; 110 01  47 copy loop template
nop0    ; 110 00  48 copy loop template
nop1    ; 110 01  49 copy loop template
nop0    ; 110 00  50 copy loop template
movii   ; 110 1a  51 move contents of [bx] to [ax] (copy one instruction)
decC    ; 110 0a  52 decrement cx (size)
ifz     ; 110 05  53 if cx == 0 perform next instruction, otherwise skip it
jmpo    ; 110 14  54 jump to template below (copy procedure exit)
nop0    ; 110 00  55 copy procedure exit complement
nop1    ; 110 01  56 copy procedure exit complement
nop0    ; 110 00  57 copy procedure exit complement
nop0    ; 110 00  58 copy procedure exit complement
incA    ; 110 08  59 increment ax (address in daughter to copy to)
incB    ; 110 09  60 increment bx (address in mother to copy from)
jmpo    ; 110 14  61 bidirectional jump to template below (copy loop)
nop0    ; 100 00  62 copy loop complement
nop1    ; 100 01  63 copy loop complement
nop0    ; 100 00  64 copy loop complement
nop1    ; 100 01  65 copy loop complement
ifz     ; 000 05  66 this is a dummy instruction to separate templates
nop1    ; 110 01  67 copy procedure exit template
nop0    ; 110 00  68 copy procedure exit template
nop1    ; 110 01  69 copy procedure exit template
nop1    ; 110 01  70 copy procedure exit template
popC    ; 110 12  71 pop cx off stack (size)
popB    ; 110 11  72 pop bx off stack (start address of mother)
popA    ; 110 10  73 pop ax off stack (start address of daughter)
ret     ; 110 17  74 return from copy procedure
nop1    ; 100 01  75 end template
nop1    ; 100 01  76 end template
nop1    ; 100 01  77 end template
nop0    ; 100 00  78 end template
ifz     ; 000 05  79 dummy instruction to separate creature
```

```
**** end genome file
```

Each genome file begins with some header information. Let me explain each item:

format: 3 because we occasionally change the format of the genome files, this parameter is included for backwards compatibility. It is used by the assembler/disassembler to know how to read and write the files. This variable is now defunct.

bits: 2156009669 this is the bit field associated with each genome in the genebank. If the genebanker is on and if any of the parameters: WatchExe, WatchMov, or WatchTem are set to a non-zero value, then bits in this field will be set to characterize the ecological characteristics of the genotype. The definitions of the bits in the field are given in the tierra.h module, and above in the description of the soup_in parameters. For more specific details, follow the Watch variables in the source modules to see exactly what they are doing.

EXsh TCsh TPs MFs MTd MBh this is an ASCII summary of the meaning of the bits that are set in the bit field. The meanings of these abbreviations are given in the tierra.h file and above in the description of the soup_in parameters.

genotype: 0080aaa This is the name of this genotype. The name has two parts. The first part is numeric and must be equal to the size of the cell of this creature (the size of its allocated block of memory). The cell size usually, but not always, corresponds to the size of the genome. The second part is a unique (and arbitrary) three letter code to distinguish this particular genotype from others of the same size.

parent genotype: 0666god This is the name of the genotype of the immediate ancestor of this genotype. The immediate ancestor is the creature, whose CPU gave rise to the first individual of this genotype. The original creature, 0080aaa was created by god and the devil. This information is deficient in three respects. First, the creature whose CPU created the offspring is not necessarily the genetic parent of the creature. Hyper-parasites for example, force other creatures to replicate their genomes. Second, the immediate ancestor of a creature may have gone extinct before it crossed the threshold, so its genotype may not appear in the genebank. Third, in the case where the immediate ancestor went extinct without being saved in the genebank, its name becomes available for reuse. This means that even if you find another creature with the right name in the genebank, there is no certainty that it is actually the ancestor you are looking for. In short, this information is essentially useless. This is the problem that is most actively being worked on at the moment. A seamless phylogeny tracker is under development.

1st_daughter: flags: 0 inst: 827 mov_daught: 80 breed_true: 1 This is a set of metabolic data about what transpired during the production of the first daughter by this genotype. flags: 0 This tells us how many errors (flags) were generated during the first reproduction. The generation of errors indicates invalid execution of instructions and causes the creature to move up the reaper queue, closer to death. inst: 827 This tells us how many instructions were executed during the first reproduction, this is an indication of metabolic costs and efficiency. mov_daught: 80 This tells us how many instructions were copied from the mother to the daughter during the first reproduction. breed_true: 1 This tells us if the first daughter ever has the same genotype as the mother.

2nd_daughter: flags: 0 inst: 809 mov_daught: 80 breed_true: 1 This is a set of metabolic data about what transpired during the production of the second daughter by this genotype. The data are the same as those from the first daughter. The second daughter and those that follow generally have the same metabolic data, but they also generally differ from the first daughter, because the second time through, the parent often does not examine itself again, and it does not start the algorithm from the same place.

Origin InstExe: 0,0 clock: 0 Thu Jan 01 -5:00:00 1970 InstExe: 0,0 At the time this genotype first appeared, the system had executed this many of instructions. The number to the left of the comma is the number of millions of instructions, and the number to the right of the comma is the remainder. clock: 0 This is the system clock time at the first origin of this genotype. Wed Dec 26 22:56:08 1990 This is the system clock time at the first origin of this genotype.

MaxPropPop: 0.8306 MaxPropInst: 0.4239 mpp_time: 0,0 MaxPropPop: 0.8306 The maximum proportion of the population of adult cells in the soup, attained by this genotype. MaxPropInst: 0.4239 The maximum proportion of space in the soup attained by adults of this genotype. mpp_time: 0,0 The time at which this genotype achived its maximum proportion of the population of cells (MaxPropPop). Time here is measured in millions of instructions (to the left of the comma) and the remainder (to the right of the comma).

ploidy: 1 track: 0 ploidy: 1 The ploidy level of this genotype (i.e., this genotype is haploid). track: 0 Which copy of the genome will start executing at birth. This is only used when the ploidy level is greater than one (i.e., diploid).

```
track 0:
```

track 0: This tells us that the assembler code that follows is track one. If the genotype has a ploidy of 2, a second assembler listing will follow, and it will be labeled track 1.

```
nop1    ; 110 01   0 beginning marker
```

This is the first line of the actual genome. The first word, nop1 is the assembler mnemonic for one of the two no-operation instructions. The semicolon indicates the beginning of comments.

The digits 110 are a record of which instructions were executed by this creature's own CPU (first digit), and the CPUs of other creatures' (second digit), the third digit is not used at present. These bits are set when the WatchExe parameter is set. That the first two digits are set to one indicates that this instruction was executed both by its own CPU and by the CPU of another creature (perhaps a parasite, or a lost instruction pointer).

The digits 01 are the actual hexadecimal op code of the instruction. It is this value that will actually be stored in the soup.

The digit 0 just before the words ``beginning marker'' is a count of the Nth instruction in the genome. This is the first instruction, so it is numbered zero.

The words ``beginning marker'' are a comment describing the intended purpose of this instruction.

If you study the code of the ancestor, you may be perplexed by the reason for including the following instructions:

```
zero    ; 110 04   4 put zero in cx
not0    ; 110 02   5 put 1 in first bit of cx
shl     ; 110 03   6 shift left cx (cx = 2)
shl     ; 110 03   7 shift left cx (cx = 4)
movDC   ; 110 18   8 move cx to dx (dx = 4)
```

In the original version of the simulator, the size of the templates was determine by the value in the dx register. These five instructions loaded the dx register with the value 4, which is the size of the templates in this creature. Later, it was decided that this was a stupid way to determine template sizes. Now the decoder just looks to see how many nops follow any instruction using them, and the number of consecutive nops determine the template size. Therefore, these five instructions don't do any useful work in the present model, but they have been left in place because the code still works.

9.2) Writing a Creature

If you write your own creature, you must obey the following conventions:

```
**** begin genome file (note blank line at top of file)

format: 3  bits: 3
genotype: 0080aaa  parent genotype: 0666god

track 0:

nop1    ;
nop1    ;
**** end genome file
```

Yank the above lines into the file you are going to write, to use as a template. You must have the following:

1) a blank line at the top of the file. 2) a line declaring the format and bits, just use the line given. 3) a line stating the genome size and three letter name, and that of the parent genotype. The genome size must match the actual number of instructions in the genome. The three letter name is arbitrary, you can make up any name, but I advise using a low-letter name like aaa because these names are used in a base 26 numbering system by the genebanker, and the genebanker must allocate an array as big as the largest of these numbers. You may make up the parent genotype size and name, it won't be used for anything, so its details don't matter, but it should have the format of four numeric digits followed by three letters. 4) a blank line 5) the line: track 0: just use the line provided 6) a blank line 7) the listing of assembler mnemonics, followed by a semicolon. The listing of the 32 assembler mnemonics can be found in the opcode.map file. For a description of what they actually do, study the comments on the code of the ancestor listed above, and study the corresponding decode and execute functions in the two modules in decode.c and instruct.c.

---

10) If you want to Modify the Source Code

If you make some significant improvements to Tierra, we would welcome receiving the source code, so that we may integrate it into our version, and then make it available to others. We will credit you for your contributions.

All lines of source code should be 78 characters or less, or it will mess up the formatting of the code for distribution by mail.

The simulator has been designed so that it can be brought down, and then brought back up where it left off. This means that there can be no static local variables. Any variables that hang around must be global. They are declared and defined in soup_in.h if they are also soup_in parameters. Otherwise they are declared in declare.h, and all global variables are declared as externals in extern.h.

The code for bringing the simulator up and down is in the tsetup.c module. The system is brought up by GetSoup(), which calls GetAVar() to read soup_in. All soup_in variables are read by the GetAVar() function. If a new simulation is being started, GetSoup() calls GetNewSoup(). If an old simulation is being restarted, GetSoup() calls GetOldSoup(). GetOldSoup() will read all global variables not contained in soup_in, and will also read in all arrays, such as the soup, the cells array, and the free_mem array. When the simulator goes down, and periodically during a run, all global variables are written to a file soup_out, and all global arrays such as soup, the cells array, the free_mem array, and the random number generator array, and some structures, are written to a binary file called core_out. Thus if you create any new global variables or arrays, be sure they are read by GetOldSoup(), and written by WriteSoup().

There are several obvious projects that I would like to comment on:

10.1) Creating a Frontend

All I/O to the console is routed through the frontend.c module, so that it can be handled by a variety of frontends now under development. The simplest of these just uses printf to write to standard out. If you compile with

#define FRONTEND BASIC, you will get the new frontend created by Dan Pirone. The Basic frontend consists of five data area: The STATS area at the top two lines of the screen. The PLAN area next for displaying simlation variable every million virtual instruction. The MESSAGE area, for state changes, and Genebank data. The ERROR area at the second to the last line of the screen. The HELP area at the last line of the screen.

If your are going to work on the frontend, please observe the Basic code as a template for changes, or get back to us for an updated version of the frontend.c module.

10.2) Creating New Instruction Sets

If you want to create a new instruction set, more power to you. The relevant modules to study are: instruct.c, decode.c, soup_in.h, arginst.h, configur.h, and opcode.map. You will also need to study the definitions of struct Cpu, struct InstDef, and type Inst, all in the tierra.h module. Note that the Cpu structure includes an array of registers. The idea is that you may change the size of this array to make just about any changes you might want to the CPU architecture. You should avoid actually having to alter the structure definitions in the tierra.h file. A great deal of reconfiguration of the instruction sets is possible at run time, by manipulating the opcode.map files.

10.3) Creating New Slicer Mechanisms

If you want to experiment with artificial rather than natural selection, consider that selection is both a carrot and a stick. The carrot in this model is CPU time which is allocated by the slicers. The stick is the reaper. If you want to try to evolve algorithms that do useful work, your evaluation functions should be embedded into the slicer, and should allocate more CPU time to creatures who rank high. You can also do the same with the reaper.

10.4) Creating a Sexual Model

Sex emerges spontaneously in runs whenever parasites appear. However, this sex is primitive and disorganized. I believe that the easiest way to engineer organized sex is to work with diploid creatures. The infrastructure to allow multiple ploidy levels is already in place. Notice that for ploidy levels higher than one, the definition of Instruction, the type of which the soup is composed is:

```
typedef struct Inst  Instruction[PLOIDY];
```

This means that if PLOIDY is defined as two, there are two parallel tracks for genomes. The instruction pointer will run down the track specified by the ce->c.a->ex variable in the Cpu structure. We implemented few other controls over the tracking of the instruction pointer in diploid or higher models. This is future work.

---

11) Configuration at Compile Time (configur.h)

The file configur.h is included by the file tierra.h, and thus is included in all tierra source code modules. Many definitions are made in this file that control options that require recompilation to change. The number of these options has been growing, therefore the following is a brief documentation of some of the parameters defined in this file.

```
#define PLOIDY 1
```

PLOIDY defines the number of parallel instruction tracks contained in the soup. Tierra is normally run as a haploid model, meaning that each creature has one copy of the genome. It is possible for creatures to contain two or more copies of the genome (humans have two copies of each chromosome), by defining ploidy to a number larger than one.

```
#define STACK_SIZE 10
```

Each of the four instruction sets includes a stack, whose size is defined by this variable.

```
#define ALOC_REG 4
#define NUMREG 4
```

Each of the four instruction sets uses an array of registers, whose size is controlled with these variables. The number of actual registers is defined by NUMREG, but there may be additional virtual registers (as in instruction set 1) which can be allocated by settin ALOC_REG larger than NUMREG.

```
#define GETBUFSIZ 3
#define PUTBUFSIZ 3
```

Instruction sets two through four include communication functions, get() and put(), which read from the get buffer and write to the put buffer. GETBUFSIZ and PUTBUFSIZ control the size of these buffers.

```
#define COMDATBUFSIZ 3
```

This definition is preparatory for having communications across the network.

```
#ifndef FRONTEND
#define FRONTEND BASIC  /* BASIC or STDIO */
#endif
```

There are two frontends. The BASIC frontend provides the best interface, but requires the use of the console. In order to run Tierra in the background (on unix systems) Tierra should be compiled with the STDIO frontend whose output can be redirected to /dev/null.

```
/* #define CM5       */ /* CM5 the version for the CM5 */
```

CM5 controls variations specific to the MIMD implementation of Tierra on the CM5, the Connection Machine manufactured by Thinking Machines Corporation.

```
#define MICRO        /* define for micro step debugging */
```

MICRO implements the code required to use the virtual debugger.

```
/* #define HSEX      */ /* define for haploid, crossover sex */
```

HSEX implements the code used to support haploid sex among creatures.

```
#define I/O          /* define for buffered input-output */
```

If the put and get instrucitons are to be used, this should be defined.

```
#define SHADOW
```

Define this varible to allow use of the option of shadow registers, as are used in instruction set one.

```
/* #define READPROT  */ /* define to implement read protection of soup */
#define WRITEPROT    /* define to implement write protection of soup */
/* #define EXECPROT  */ /* define to implement execute protection of soup */
```

Instructions in the soup can be protected using the mal() instruction. The status of protected and unprotected memory is determined by the two soup_in variables:

```
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  read, write, execute protection for my memory
MemModeProt = 2  read, write, execute protection for other creature's memory
```

However, each of the three memory protection modes costs CPU time, therefore they should not be compiled unless they are going to be used. The READPROT, WRITEPROT, and EXECPROT definitions control which protection modes are implemented at compile time.

```
/* #define ERROR     */ /* use to include error checking code */
```

ERROR turns on a large body of error checking and verification code. It should only be used while debugging Tierra.

```
/* #define ALCOMM    */ /* define for socket communications */
```

ALCOMM turns on the code used to communicate with the ALmond monitor program (unix only).

```
/* #define MEM_PROF  */ /* profile dynamic memory usage */
```

MEM_PROF turns on code that provides a summary of how dynamically allocated memory is being used. This is basically for debugging purposes.

12) Creating a Multi-cellular Model

Multi-cellularity was the hallmark of the Cambrian explosion of diversity, and thus is likely a biological feature worth including in Tierra. Also, it is likely that a multi-cellular model is the appropriate one for evolving large application programs on massively parallel machines. How can we implement multi-cellularity? What does it mean in the context of Tierran creatures?

Multi-cellularity without cell differentiation is rather uninteresting. Consider that at the conceptual core, cell differentiation means that the mother cell determines what portion of the genome its daughter cell will express. For many daughter cells, the mother cell narrows their options by preventing them from expressing (executing) large portions of their genome (code). In the organic world this is done by loading the daughter cell with regulatory proteins which determine which genes will be expressed.

In the Tierran world, the same result can be achieved by allowing the mother cell to set the position of the instruction pointer in the daughter cell, and also the initial values of the CPU registers. These acts can place the daughter cell into a portion of its code from which it may never be able to reach certain other parts of its code. In this way the mother cell determines what parts of the code are executed by the daughter.

To facilitate this process, the divide instruction has been broken into three steps: 1) Create and initialize a CPU for the daughter. 2) Start the daughter CPU running. 3) Become independent from the daughter by loosing write privelages on the daughter space. Now, between steps 1 and 2, the mother can place values into the CPU registers and instruction pointer of the daughter. This will require and inter-CPU move instruction. The divide instruction takes an argument that determines which of the three steps is being performed. The divide instruction in the new instruction sets does all three steps at once, but it also transfers the value from the mother's CPU registers to the daughter's CPU registers, and it starts the daughter's instruction pointer at some offset into the daughter's genome, as specified by a value in one of the mothers CPU registers (this can cause differentiation).

Transfer of the concept of multi-cellularity from the organic to the digital domain could take many forms. To make the transfer we must first understand what the most basic, essential, and universal features of multi-cellularity are, and then find the form that these feature would take in the completely different physics of the computational system into which evolution is being introduced. Version 4.1 of Tierra includes an initial attempt at implementing multi-cellularity, in which we aim to capture the following features:

1) That multi-cellular organisms originate as single cells, which develop into multi-celled forms through a process of binary cell division.

2) That each cell of a multi-celled individual has the same genetic material as the original cell from which the whole developed.

3) That the different cells of the fully developed form have the potential for differentiation, in the sense that they can express different parts of the genome (they could execute different parts of the program).

In the digital metaphor of multi-cellularity, the program is the genome, and the processor corresponds to the cell. In organic biology, there is at least one copy of the genome for each cell, because genetic information can not easily be shared across cell membranes. In most parallel machines, the same holds: there is an area of memory associated with each processor (cell), and there must be at least one copy of program code in the memory of each processor, because the memory is not shared. This provides a very neat model of multi-cellularity: each digital cell consists of a unique block of memory with its own copy of the program and it's own processor.

However, if the parallel machine is a shared memory machine, making copies of the genome for each cell is wastefull of memory and processing time (for copying the genetic information), and is simply not necessary. In this context it is highly unlikely that evolution by natural selection would find any advantage in such waste. Thus the logical and evolutionarily more efficient implementation in this context is that a single copy of the program in a single block of memory shares multiple processors. The different cells then correspond only to the different processors. These may still develop from a single original processor through splitting. They may also exhibit differentiation by executing different parts of the program. And obviously all cells will contain the same genetic material, since there will actually be only one copy per multi-cellular individual. The ability in version 4.1, of single cells to have more than one processor provides an implementation of these ideas.

Even in V4.1, creatures are born with a single processor. In order to spawn additional processors, they must execute the split() instruction. The split instruction creates a processor identical to the mother processor, except that the value in the dx register will be manipulated so that the processors can be differentiated. In addition, there is a join() instruction that causes all the processors in a single cell to colapse into one after all of the processors have executed the join() instruction. There is also a csync() instruction that can allow the processors to synchronize by lining up at the csync() instruction.

---

13) If you have Problems

13.1) Problems with Installation

Read the installation instructions carefully. Where we present alternative methods, try them all. If it still doesn't work, please let us know. We would like to help people with installation, and make sure that our instructions are clear. If you are using a compiler or machine we have not tested, we may not be able to help you, but we want to accomodate these additional conditions. We would like to help you find a solution and incorporate the solution in future releases.

13.2) Problems Running Tierra

Read all of the .doc files. You may find the answer there. If a problem still persists, and you have ftp access, get a new copy of the source code out of the ftp site. It is likely that the source code will be updated from time to time as we continue to improve the program. By the time you are sure there is a problem, we may already have fixed it and placed a fix in the ftp site.

If the problem still persists after you have tested the latest version of the software, let us know about the problem. We would like to fix it. If you do not have ftp access, and you identify a bug, we will fix it and send you a free upgrade if you return you original distribution disk in a disk mailer.

---

14) Tierra ftp Sites

The Tierra V4.1 source code; and the source code, and DOS executables of all tools; and documentation and manuscripts is available. Please note that the source code in the ftp site and the source code provided on disk will each compile and run on either DOS or UNIX platforms. It is exactly the same source code in either case. The DOS executables are available only on disk (available from Virtual Life), and can not be freely distributed.

The complete source code and documentation (but not executables) is available by anonymous ftp at:

tierra.slhs.udel.edu [128.175.41.34] and life.slhs.udel.edu [128.175.41.33]

the file: tierra/tierra.tar.Z

To get it, ftp to tierra or life, log in as user ``anonymous'' and give your ema il address (eg. tom@udel.edu) as a password. Change to the tierra directory and get tierra.tar.Z, a compressed tar file. Be sure to transfer in binary mode. It will expand into the complete directory structure with the following commands (Unix only):

uncompress tierra.tar.Z tar oxvf tierra.tar

The contents of the ftp site described above are mirrored in another ftp site: alife.santafe.edu in the directory: pub/SOFTWARE/Tierra

---

15) Registration & Mailing Lists

The reason you might want to register your copy of the software is so that we can contact you if we discover a bug, or we can let you know when new versions of the program are ready for distribution. If you obtained Tierra on disk, send your name, address and the serial number of your distribution disk to Virtual Life at the address listed below. If you obtained Tierra from the ftp site, send your email address to: ray@santafe.edu

tierra-announce official updates, patches and announcements only

The addresses are:

ray@santafe.edu the list administrator (Tom Ray). to be added, removed, or complain about problems with any of these lists.

tierra-bug@life.slhs.udel.edu for bug-reports or questions about the code or installation.

There is a new Artificial Life news group: comp.ai.alife

---

16) Artificial Life Online

The following announcement was distributed by Chris Langton on April 22, 1994. The announcement is outdated now, and the service has been changed drastically, but you can get some info at some of the addresses listed.

ANNOUNCING:

ARTIFICIAL LIFE ONLINE

The Artificial Life Online WWW-Server and BBS Service

Sponsored by

MIT Press and The Santa Fe Institute

alife.santafe.edu

The Artificial Life Online/BBS is intended to be a central information collection and distribution site on the Internet for any and all aspects of the Artificial Life endeavor. The system is sponsored by MIT Press and the Santa Fe Institute.

The Alife Online service combines the functionalities of a WWW server, a Gopher server, an FTP site, an interactive bulletin-board-system, and Usenet News. Directions for accessing Alife Online and the ALBBS in these different modes are included below.

A special feature is a collection of 40 or so local newsgroups dedicated to a wide variety of topics in Artificial Life.

Many of the files and resources here are available to everybody via Gopher and WWW. However, to access the full range of BBS services, it is necessary to come in using telnet and to create a local account. This will allow you to participate in the local Alife newsgroup discussions, and to set up personal information files such as a plan, project, HTML personal home page, etc.

To access Alife Online via World-Wide-Web (WWW):

Use the URL http://alife.santafe.edu/

For best results we suggest using a client capable of handling color graphics and forms, such as Mosaic.

A character-based (ASCII) client called "lynx" is also available -- but will not support graphics.

To access the Alife Online BBS (ALBBS) via telnet:

telnet to "alife.santafe.edu" and login as "bbs". You will find yourself in a specially constructed UNIX shell within which either BBS menu commands or UNIX commands can be used to browse around in the system.

To set up a local account, telnet to "alife.santafe.edu" login as "bbs," and run the "account" program. These accounts will initially be provided free of charge, but we will eventually have to charge a nominal fee in order to cover operating expenses (on the order of $15-$25 per year). Subscribers to the Artificial Life Journal from MIT Press will have this fee waived.

Once you have an account on alife.santafe.edu, you can telnet to "alife.santafe.edu" and login as yourself.

You do not have to create an account to use the ALBBS via telnet - you can simply login as "bbs" and browse through the system using the BBS commands.

To access the www features in the context of a character based client, telnet to alife.santafe.edu and login to the BBS as "lynx".

To access Artificial Life Online using Gopher:

Connect to alife.santafe.edu (standard gopher port 70).

To access Artificial Life Online via FTP:

ftp to alife.santafe.edu, login as "anonymous" and type your login@homesite as the password.

Everything interesting is in the "pub" directory.

Feedback:

Please let us know if you have any suggestions or questions about the Alife Online/BBS system.

Send Email to:

feedback@alife.santafe.edu

Tom Ray

Zoology Department
University of Oklahoma
Norman, Oklahoma 73019

or

Santa Fe Institute
1399 Hyde Park Road
Santa Fe, NM 87501

ray@santafe.edu (email)
505-984-8800 (Phone)
505-982-0565 (Fax)

or

ATR Human Information Processing Laboratories
2-2 Hikaridai
Seika-cho Soraku-gun
Kyoto 619-02 Japan

ray@hip.atr.co.jp (email)
(81)-7749-5-1063 (Phone)
(81)-7749-5-1008 (Fax)